HUMBOLDT-UNIVERSITÄT ZU BERLIN


# Model Transformation Languages for Domain-Specific Workbenches

**DISSERTATION**

zur Erlangung des akademischen Grades

doctor rerum naturalium (Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
Humboldt-Universität zu Berlin

von

**Arif Wider**

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Jan-Hendrik Olbertz

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät:
Prof. Dr. Elmar Kulke

Gutachter:

1. Prof. Dr. Joachim Fischer, Humboldt-Universität zu Berlin
2. Prof. Dr. Andreas Prinz, University of Agder, Norway
3. Prof. Dr. Krzysztof Czarnecki, University of Waterloo, Canada

**eingereicht am:** 2. September 2014

**Tag der Verteidigung:** 18. November 2015

**Abstract**

*Domain-specific languages* (DSLs) are software languages which are tailored to a specific application domain. DSLs enable domain experts to create domain-specific *models*, that is, high-level descriptions of domain knowledge. As any other software languages, DSLs rely on *language tools* which provide assistance for processing and managing domain-specific models. A *domain-specific workbench* is an integrated set of such tools for a DSL. A recently proposed approach is to automatically generate a domain-specific workbench for a DSL from a description of that DSL. However, existing tools which apply this approach do not support to describe and generate editable domain-specific *views*. A view is a part of domain-specific workbench that presents only one aspect of a model, for example, its hierarchical structure.

This dissertation presents special *model transformation languages* which support the description of view synchronization in a generated domain-specific workbench. This allows a multi-view domain-specific workbench to be created with existing tools for language tool generation.

We present a generated domain-specific workbench for the nanophysics domain and present a taxonomy of synchronization types. This allows us to precisely define what model transformations are required for view synchronization in that workbench. According to these requirements, we develop two transformation languages by adapting existing ones. In particular, we develop a *bidirectional transformation language*. With such a language one can describe a relation which defines whether two models are in sync and let the synchronization logic be inferred automatically. We implement model transformation languages as *internal DSLs* – that is, embedded as expressive libraries – in the *Scala* programming language and use Scala's type checking for static verification of transformations and their composition.

**Zusammenfassung**

*Domänenspezifische Sprachen* (engl. domain-specific languages, DSLs) sind Software-Sprachen, die speziell für eine bestimmte Anwendungsdomäne entwickelt wurden. Mithilfe von DSLs können Domänenexperten ihr Domänenwissen auf einem hohen Abstraktionsniveau beschreiben und so domänenspezifische *Modelle* erstellen. Wie andere Software-Sprachen auch, benötigen DSLs *Sprachwerkzeuge*, die Assistenz bei der Erstellung und Verarbeitung von domänenspezifischen Modellen bieten. Eine *domänenspezifische Werkbank* ist ein Software-Werkzeug, welches mehrere solcher Sprachwerkzeuge für eine DSL miteinander integriert. In den letzten Jahren wurde ein Ansatz entwickelt, der es erlaubt eine domänenspezifische Werkbank aufgrund der Beschreibung einer DSL automatisch generieren zu lassen. Existierende Werkzeuge, die diesen Ansatz anwenden, unterstützen jedoch nicht die Beschreibung und Generierung von editierbaren domänenspezifischen *Sichten*. Eine Sicht ist ein Teil einer domänenspezifischen Werkbank, der nur einen bestimmten Aspekt eines Modells darstellt, beispielsweise dessen hierarchische Struktur.

Diese Dissertation stellt spezielle *Modelltransformationssprachen* vor, mit denen die Synchronisation von Sichten in einer generierten domänenspezifischen Werkbank beschrieben werden kann. Dadurch können domänenspezifische Werkbänke mit editierbaren Sichten mittels existierender Werkzeuge zur Generierung von Sprachwerkzeugen erstellt werden.

Dafür wird eine domänenspezifische Werkbank für die Nanophysik-Domäne sowie eine Taxonomie von Synchronisationstypen vorgestellt, welche es erlaubt genau zu bestimmen, welche Art von Modelltransformationen für die Synchronisation von Sichten in dieser Werkbank benötigt werden. Entsprechend dieser Anforderungen werden zwei Modelltransformationssprachen entwickelt, und zwar indem existierende Transformationssprachen entsprechend angepasst werden. Insbesondere wird eine *bidirektionale Transformationssprache* entwickelt. Mit solch einer Sprache kann man eine Relation, welche definiert ob zwei Modelle synchron sind, so beschreiben, dass die entsprechende Synchronisationslogik automatisch abgeleitet werden kann. Die gezeigten Modelltransformationssprachen werden als *interne DSLs* – das heißt eingebettet als ausdrucksstarke Bibliotheken – in der Programmiersprache Scala implementiert. Auf diese Weise kann Scalas Typprüfung genutzt werden, um Transformationen und deren Komposition statisch zu verifizieren.

# Acknowledgements

First and foremost I want to thank my supervisor Prof. Achim Fischer for his trust, his continuous support, and for giving me the freedom to pick my research topic and my approach as I saw fit.

Furthermore, I am grateful to all the people who supported and inspired me over the years: Prof. Ulrich Grude and Prof. Sebastian von Klinski from my Alma Mater TFH Berlin for encouraging me to pursue a PhD in the first place; my parents for always encouraging me and for their unrestricted support; my colleagues at graduate training group METRIK, in particular Daniel Sadilek and Guido Wachsmuth for helping me to identify my topic, and Siamak Haschemi and Markus Scheidgen for their continuous discussions, feedback, and co-authoring efforts; Frank Kühnlenz, Michael Frey, Andreas Reimer, Dirk Fahland, Sebastian Heglmeier, Stephan Weißleder, Michael Soden, Joanna Geibig, Artin Avanes, Christoph Wagner, Jens Nachtigall, Andreas Dittrich, Jan Calta, Hartmut Lackner, Björn Lichtblau, Matthias Sax, Christian Blum, and Andreas Blunk for their company and solidarity, in particular at numerous METRIK evaluation workshops; Martin Schmidt and Lars George whose master theses I was happy to supervise and who are now PhD candidates themselves - without them this work would not have been possible; Michael Barth, Janik Wolters, and Prof. Benson from the nano-optics group for the great cooperation; the professors of METRIK for their valuable feedback, in particular Prof. Holger Schlingloff for his co-supervision and for welcoming me in his research group; Zinovy Diskin for early inspiration and many fruitful discussions a few years later; Prof. Eelco Visser, Prof. Andreas Prinz, Prof. Krzysztof Czarnecki, and Prof. Zhenjiang Hu for welcoming me at their institutions for research visits; the DFG and the DAAD for their financial and organisational support; Gabriele Graichen, Marita Albrecht, Manfred Hagen, and Silvia Schoch for their continuous administrative support; Tony Crawford for valuable corrections and suggestions regarding my writing; the various coffee shops where I wrote large parts of this dissertation, in particular Niels and Malte from Leuchtstoff Kaffeebar, Kamee, Thor, Oslo kaffebar, Westberlin, Five Elephant, and No Fire No Glory.

Finally, and most importantly I want to thank my lovely wife Olivera, not only for her invaluable support over the years but also for the countless hours of actively helping me understanding my research problems and helping me with the completion of this dissertation.

# Contents

# 1 Introduction

This dissertation contributes to the interrelated research fields of model-driven engineering (MDE) and software language engineering (SLE). MDE is a methodology in software engineering which is concerned with generating software from high-level descriptions called *models*. Central elements of MDE therefore include special languages for describing models, called *modeling languages*, and special languages for describing the transformation from models to software, called *model transformation languages* (Schmidt, 2006). Model transformation languages and modeling languages are *software languages*: like programming languages, they are non-natural languages which are intended to be processed by a computer. SLE is concerned with the development of software languages and with the development of software to process such languages, called software language *tooling*. More specifically, model transformation languages and most modeling languages are *domain-specific languages* (DSLs). A DSL is a language which is tailored to a specific application domain (Fowler, 2010). An example of a technical DSL is SQL, which is tailored to the task of concisely expressing database queries.

With the advent of integrated development environments (IDEs) such as *Eclipse*, language tooling for popular programming languages like Java has become increasingly rich-featured, and provides extensive assistance to users of the language, including error highlighting, quick fixes, code navigation, and refactoring support. As a result, user expectations of software language tooling in general have increased. This can be a problem for DSLs. Because of their narrow application domain, many DSLs have a smaller user base than general-purpose programming languages such as Java or general-purpose modeling languages such as UML. This makes it difficult to justify the high costs of developing rich-featured tooling for a DSL (Völter et al., 2013). However, because of increased expectations, a lack of powerful language tooling can inhibit a DSL's success.

To alleviate this situation, a recently proposed approach applies MDE to the development of language tooling: that is, to generate language tooling from models that describe a language and its tooling (Nytun et al., 2006; Scheidgen, 2008; Heidenreich et al., 2013). One tool that applies this approach is *Xtext* (Efftinge and Völter, 2006). *Xtext* generates a rich-featured editor for a textual language from a description of the language's grammar. Fowler (2005) has coined the term *language workbench* for a tool that allows a language to be described and creates rich-featured language tooling from this description. A language workbench provides special *meta-languages*, that is, languages for describing languages. A *domain-specific workbench* is an integrated set of tools for a DSL or several DSLs specific to the same domain and can be created using a language workbench[1].

[1] We will explain the special terms used so far – model, modeling language, DSL, domain-specific workbench, etc. – in more detail, and define their precise usage in this dissertation, in Chap. 2, Foundations.

## 1.1 View Synchronization in Generated Language Tooling

When we developed a DSL and a corresponding domain-specific workbench for a subdomain of nanophysics using *Xtext*, we found that it was difficult to add *multi-view editing* capabilities to the workbench generated. However, multi-view editing has already become common in Java language tooling to some extent. An example is the *outline view* provided by the *Eclipse Java Development Tools*[2] (JDT). An outline view is a typical user interface element of today's language tooling. In programming language tooling, the outline view usually visualizes the hierarchical structure of syntax elements, such as classes and their members, in the source code file currently opened (Fig. 1.1).

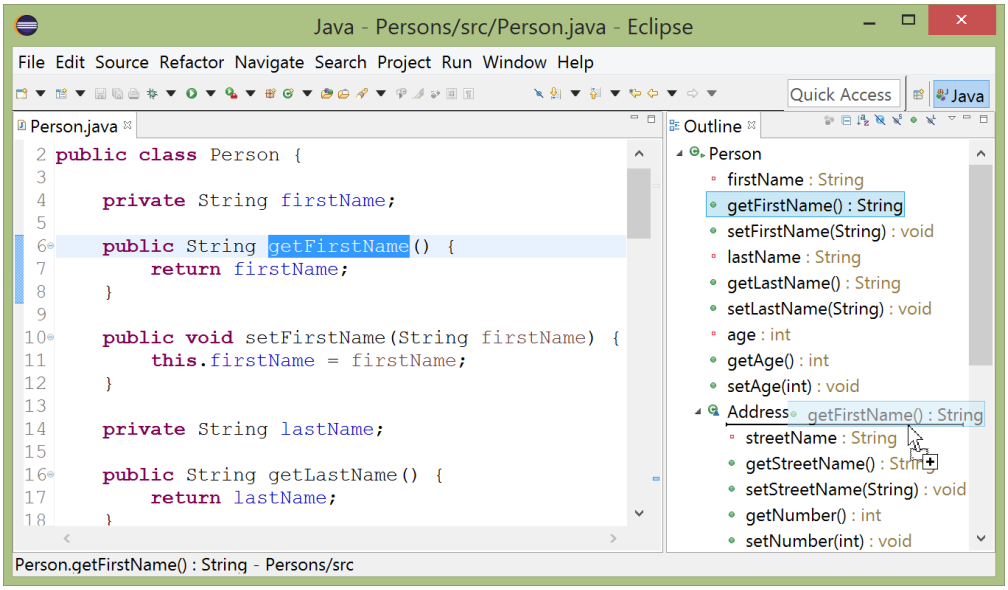



Figure 1.1: At the right, the outline view provided by the *Eclipse Java Development Tools*

The purpose of an outline view is to provide a quick overview of a file's contents. An outline view therefore presents only selected information from the code file, omitting details such as method bodies for example. This is consistent with the meaning of *view* in database theory: a view is the result set of a query, and often represents a subset of the data contained in a table (Atzeni and Torlone, 1996). Similarly, we will use the term 'view' in this dissertation to refer to a user interface element that displays selected information retrieved from some larger data source. Now, the JDT outline view is an *editable view*: that means it does not merely present data, but also allows certain edits to be made and to be propagated back to the data source – in this case, the code file. For example, class members can be moved within the hierarchy (as shown in Fig. 1.1 with the `getFirstName` method), their order can be changed, or they can be deleted directly in the outline. In each case, the code is modified accordingly. Of course, edits made to the code file in the main textual Java editor are also immediately reflected in the outline view. This is what we call *view synchronization*.

[2] http://eclipse.org/jdt

Manually implementing a *domain-specific* editable view – that is, not just a generic hierarchical outline view, but a view presenting domain-specific visualizations – is costly and brings with it the same problem mentioned in the opening of this chapter with regard to DSL tooling in general. It may be be beneficial to use the same approach of generating language tooling, and the same language workbench technologies, to create domain-specific views – in other words, to describe a view as one would describe a language, then automatically generate the user interface code for the view from that description. However, the data presented by such a generated view still needs to be synchronized with the data presented by other parts of a domain-specific workbench, such as a generated DSL editor. Now, if a view is created in similar way as a DSL editor, then view synchronization can be described as the synchronization of utterances of different languages (Garcia, 2008; Kalnina and Kalnins, 2008). Fig. 1.2 illustrates this approach to view synchronization. We will describe it in more detail and compare it with other approaches in Sec. 3.1.5.

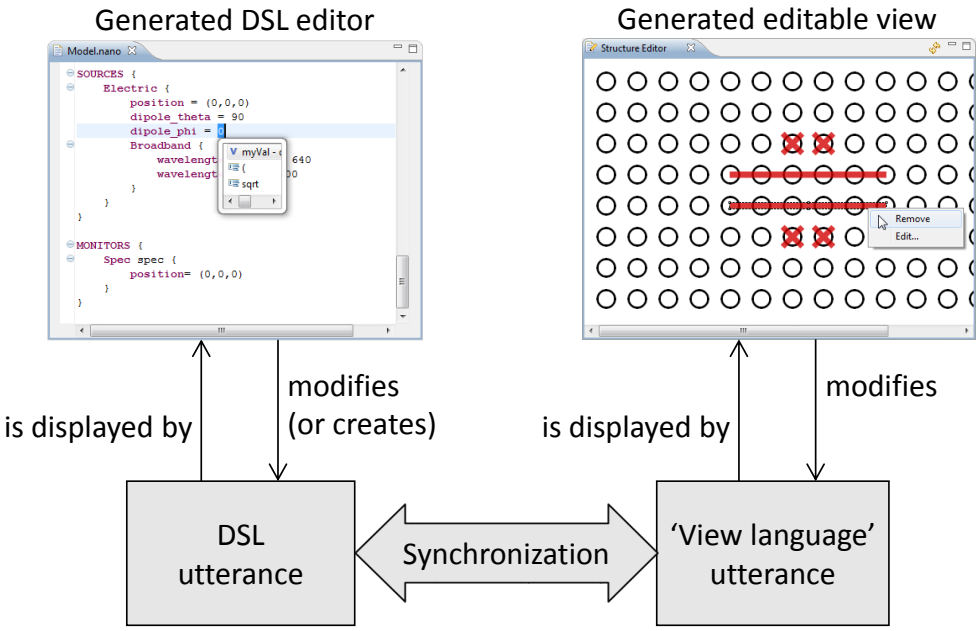



Figure 1.2: Synchronizing a generated view by synchronization of language utterances

In the example illustrated, the initial synchronization of the view can be achieved by transforming an utterance created by the DSL editor into a corresponding utterance of the 'view language'. We call this the forward transformation. When edits are made in the view, they can be propagated back by transforming the modified view utterance into a corresponding DSL utterance and then either updating the original DSL utterance accordingly or replacing it with the updated one. We call this the backward transformation.

However, implementing a synchronization by separately describing a forward and a backward transformation has several disadvantages. First, it must be ensured that the two transformations are – broadly speaking – each other's inverse (Matsuda et al., 2007; Hettel et al., 2008; Stevens, 2008). If the transformations are described using a general-purpose

programming language, it may be undecidable whether that is the case. Second, the two transformations must be maintained separately: in other words, if the view's specification is changed, both transformations need to be manually changed accordingly. To avoid this redundant effort, it is desirable to concisely describe one consistency relation which defines whether two utterances of two different languages correspond to each other and let the two transformations be inferred automatically. This is the purpose of *bidirectional transformation languages* (Stevens, 2007b; Czarnecki et al., 2009).

## 1.2 Problem Statement

The approach to view synchronization presented in Sec. 1.1 could be used to create a multi-view domain-specific workbench, with little manual implementation effort, by describing languages, views, and synchronizations – using suitable languages or meta-languages for each task – and then generating large parts of the workbench using existing language workbench technology.

The problem with this approach is the lack of suitable languages for describing synchronizations. At the time[3] we started with the development of the nanophysics workbench mentioned in Sec. 1.1,

> *the existing transformation languages either did not allow the concise description of those kind of synchronizations required for view synchronization in a generated domain-specific workbench or could not be integrated with existing language workbench technologies without significant effort.*

In regard to the first part of this problem, an important requirement (but not the only one) is support for *non-bijective* synchronization. In a bijective synchronization, every element in a data set matches exactly one corresponding element in the other data set with which it is to be synchronized (Stevens, 2007b; Antkiewicz and Czarnecki, 2007). Because a view presents only selected information from a data source, bijectivity rarely occurs in view synchronization. In Secs. 3.2 and 3.3, we precisely define requirements for suitable transformation languages by presenting a taxonomy of synchronization types and by identifying the required types.

In regard to the second part of the problem, there are transformation languages which allow the description of non-bijective synchronization but cannot be used in conjunction with technologies such as *Xtext* because these existing languages belong to a different *technological space*.

*Definition* 1.1 (technological space). *"A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. It is often associated to a given user community with shared know-how, educational support, common literature and even workshop and conference meetings."* (Kurtev et al., 2002)

[3] Late 2009

Often, a solution developed in one technological space cannot be applied in a different technological space due to conceptual or technological gaps, although the corresponding problem is basically the same (Wimmer and Kramler, 2005). *Xtext* and several other language workbench technologies for creating a domain-specific workbench belong to the *modelware* technological space.

*Definition* 1.2 (modelware). *Modelware* is the technological space of model-driven engineering. It is characterized by object-oriented concepts and the *Meta-Object Facility* (MOF) meta-modeling standard (Bézivin, 2006).

In the modelware technological space, Java-based technologies – especially the *Eclipse Modeling Framework*[4] (EMF) – also play an important role (Atkinson and Kühne, 2003; Clark et al., 2008). Technically all language utterances created by EMF-based language tooling such as *Xtext* are models[5]. Therefore, view synchronization in a generated EMF-based domain-specific workbench is actually *model synchronization* and can only be implemented using (ideally bidirectional) *model transformation* languages, in other words, transformation languages which belong to the modelware technological space.

*Focal*, for instance, is a bidirectional tree transformation language which supports non-bijective synchronization (Foster et al., 2007). *Focal* applies a compositional approach where small transformations, called *lenses*, are used to compose more complex transformations. However, because it was developed in the technological space of functional programming (sometimes referred to as *lambdaware*), with *Focal* one cannot directly transform models which were created by modelware language tools. We will explain the conceptual and technological challenges in more detail in Chap. 5. *GRoundTram* is another example of a transformation tool which supports non-bijective synchronizations but cannot be applied – at least not seamlessly – to the view synchronization scenario presented in Sec. 1.1 because of its functional origin (Hidaka et al., 2011).

On the other hand, there are model transformation languages which could be applied technologically but do not meet the specific requirements for concisely describing view synchronization. For example, *QVT-Relations*[6] is a bidirectional model transformation language which supports the declarative description of model synchronization but does not provide clear semantics for non-bijective synchronization (Stevens, 2010). These semantic issues might also be the reason why there is no actively maintained tool support for *QVT-Relations*. The *Atlas Transformation Language* (ATL, see Bézivin et al., 2003; Jouault et al., 2008) is a model transformation language with clear semantics and good tool support but does not support the definition of bidirectional transformations, which means a forward and a backward transformation must be specified separately which imposes the maintenance and invertibility issues we outlined earlier.

[4] http://eclipse.org/modeling/emf
[5] we will explain the technical and the conceptual meaning of 'model' in Chap. 2, Foundations.
[6] http://omg.org/spec/QVT/

## 1.3 Approach

Summing up, there are existing transformation languages which meet the conceptual requirements to concisely describe view synchronization but cannot be used effortlessly in conjunction with language workbench technologies from the modelware technological space. Our aim therefore is to accomplish a knowledge transfer between technological spaces by adapting these languages in such a way that they are applicable in the modelware technological space.

Our concrete approach for this is to

> *create model transformation languages which seamlessly integrate with modelware technologies by implementing existing transformation languages as internal DSLs in the Scala programming language.*

An *internal DSL* is easiest explained as a software library implemented in another software language, the *host language*, in such a way that using the library feels like using a specially tailored DSL (Fowler, 2010). An internal DSL (also called *embedded DSL*) is usually contrasted with an *external DSL* (also called *independent DSL*) which comes with its own tools, for example, a compiler. The main advantage of the internal DSL approach over the external DSL approach is that an internal DSL can reuse parts of its host language and – importantly – all of the host language's tooling. The internal DSL approach can be seen as an alternative to the aforementioned approach of generating language tools from language descriptions because the two approaches avoid manual implementation of DSL-specific tooling. We will formally define what an internal DSL is and compare the approach with the external DSL approach in Secs. 2.2.5 and 4.1.2.

Our reasons for choosing Scala[7] as the host language can be summarized as follows: Scala is a language which is based on the Java platform. This enables seamless integration with Java-based technologies such as EMF and *Xtext*. Additionally, Scala combines object-oriented and functional concepts. This is helpful for implementing concepts which originate from functional programming in an object-oriented modelware setting. Furthermore, Scala has a static type system. This allows us to provide language tooling with advanced user assistance. Finally, Scala provides features which make it particularly suitable to create internal DSLs (Sloane, 2008; Pointner, 2010). In Sec. 4.1.3, we will compare Scala with other potential host languages in order to make our decision transparent.

## 1.4 Hypothesis and Assumptions

The hypothesis of this dissertation is that

> *model transformation languages which allow the implementation of non-bijective model synchronization as required in generated multi-view domain-specific workbenches built from unmodified modelware language workbench technologies can be implemented as internal DSLs.*

[7] http://scala-lang.org

The aim of this dissertation is to show this hypothesis. Our hypothesis, however, relies on assumptions whose treatment is beyond the scope of this dissertation. Our main assumptions are the following:

1. Using and combining modelware language workbench tools, with the help of model synchronization, is beneficial for creating multi-view domain-specific workbenches.
2. The achievable usability of model transformation languages which are implemented as internal Scala DSLs is acceptable for developers of domain-specific workbenches.

It is not the aim of this dissertation to show these assumptions. In particular, we do not try to assess these assumptions with empirical studies. We show the general applicability of our approach. We also show that the internal model transformation languages which we develop are similarly expressive as existing external model transformation languages. Designing and carrying out empirical studies to assess our assumptions is the next logical step after this dissertation and is left for future work.

## 1.5 Contributions and Structure

To show the hypothesis, we specify what transformation languages are required – both conceptually and technologically – for model synchronization in a domain-specific workbench and then, according to these requirements, develop two model transformation languages. The following original contributions are presented in this dissertation:

*C1* A textual DSL for describing experiments in simulation-driven development of optical nanostructures (*NanoDSL*), and a corresponding domain-specific workbench (*NanoWorkbench*) which serves as the motivational background for this dissertation and allows us to derive requirements for suitable model transformation languages.

*C2* A taxonomy of model synchronization types that allows us to precisely define the conceptual requirements of a given model synchronization scenario.

*C3* An approach to implement type-safe model transformation languages which seamlessly integrate with EMF-based technologies as internal DSLs in Scala.

*C4* A rule-based model transformation language for unidirectional transformations based on ATL, implemented as an internal Scala DSL.

*C5* An approach to the conceptual adaptation of the *Focal* tree transformation language for model transformation.

*C6* A compositional bidirectional model transformation language based on *Focal*, that uses type-level computation for static verification of lens composition.

We provide *prototypical implementations* for contributions *C1*, *C4*, and *C6*, and report about a *case study* which demonstrates that the presented model transformation languages are suitable for implementing practical model transformation and view synchronization tasks in a domain-specific workbench. The remainder of this dissertation is organized as follows:

- In Chap. 2, we define our terminology and present the foundations of our work. We also give a brief overview of the Scala syntax.
- In Chap. 3, we present the *NanoDSL* and the *NanoWorkbench* (*C1*), discuss view synchronization, build the taxonomy of model synchronization types (*C2*), and – based on the *NanoWorkbench* and the taxonomy – specify the requirements for suitable model transformation languages.
- In Chap. 4, we present our approach of implementing model transformations as internal DSLs in Scala (*C3*), apply this approach to the development of a unidirectional model transformation language (*C4*), and discuss how this language benefits from static type checking.
- In Chap. 5, we explain the concept of *lenses*, which *Focal* is based on, present our approach to the conceptual adaption of *Focal* for model transformation (*C5*), and apply this approach to the development of a bidirectional transformation language (*C6*).
- In Chap. 6, we present a case study where we apply the two model transformation languages to practical model transformation and view synchronization tasks in the *NanoWorkbench*.
- We conclude the dissertation in Chap. 7.

Fig. 1.3 illustrates which contributions are presented in which chapter, and where important topics are discussed and related terminology is defined. Most chapters build on one another. For instance, the bidirectional transformation language which we develop in Chap. 5 uses concepts for internal DSL development in Scala that are introduced in Chap. 4. This is one of the reasons why we first present a unidirectional transformation language although our ultimate goal is to create a bidirectional transformation language.

## Acknowledgements

This dissertation partly describes work performed in cooperation with different colleagues and portions of it are based on papers written in collaboration with them. In particular, the presentation of the *NanoDSL* and the *NanoWorkbench* in Sec. 3.1 is based on an article by Wider, Schmidt, Kühnlenz, and Fischer (2011) and contains material from a corresponding master thesis by Schmidt (2011) which has been supervised by the author of this dissertation. The taxonomy of synchronization types (*C2*) presented in Sec. 3.2 is based on an article by Diskin, Wider, Gholizadeh, and Czarnecki (2014). The presentation of the unidirectional transformation language in Secs. 4.2 and 4.3 is based on an article by George, Wider, and Scheidgen (2012) and contains material from a corresponding master thesis by George (2012) which has also been supervised by the author of this dissertation.

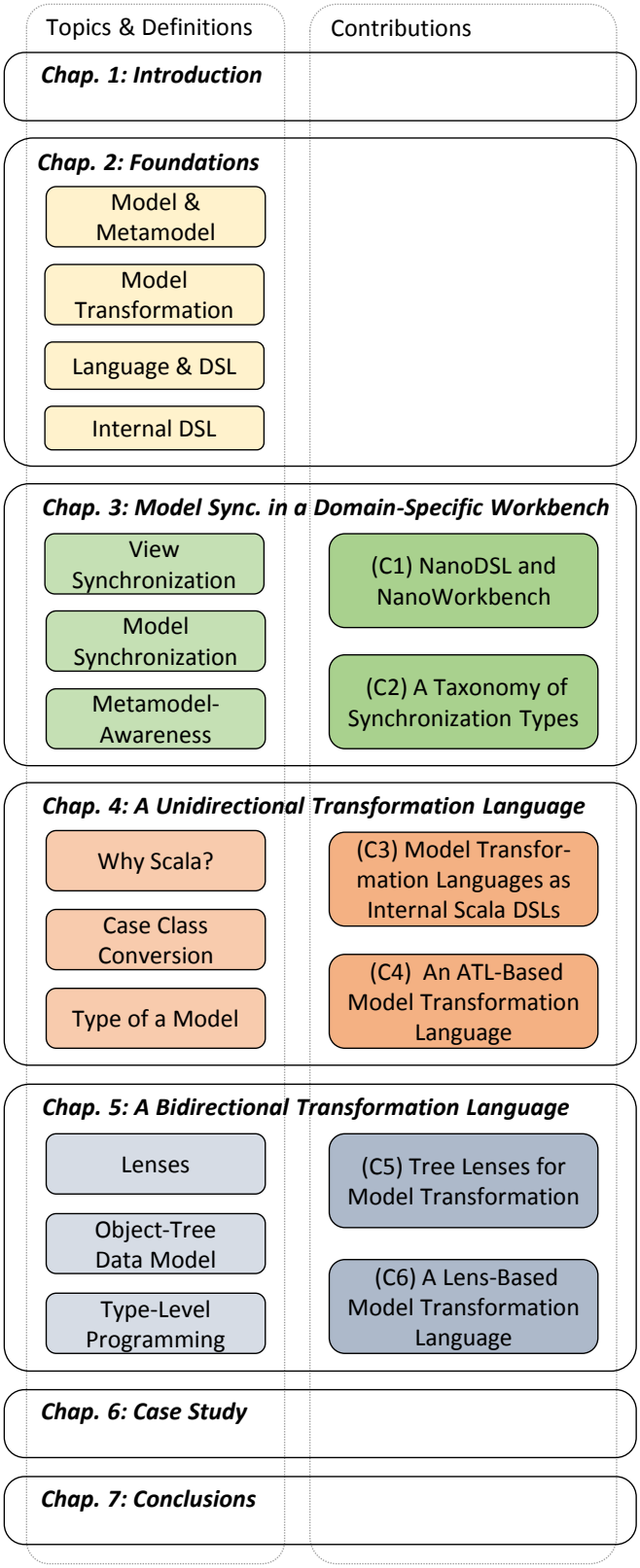



Figure 1.3: Overview of topics and contributions in this dissertation

# 2 Foundations

Terminology in model-driven engineering (MDE) and software language engineering (SLE) is still evolving and is not always used consistently in the literature. In this chapter we establish a consistent set of concepts and terminology that serves as the foundation for the subsequent contribution chapters. We assume that the reader has knowledge about basic set and graph theory, grammars, UML class diagrams, and Java.

This chapter consists of two parts: a conceptual part and a (shorter) technical part. The conceptual part covers the conceptual foundations of MDE and SLE. In the latter we also clarify the terms 'modeling language', 'programming language', 'domain-specific language', etc. The technical part briefly introduces those concepts of the *Eclipse Modeling Framework* (EMF) and of the Scala programming language which are relevant for the approaches and implementations that we present in this dissertation. Notably, in Sec. 2.3.1 we explain our important assumption that EMF-based models contain a spanning tree.

## 2.1 Model-Driven Engineering

MDE is concerned with *modeling*, that is, with creating models, and with processing models. A model in MDE is, generally, a description of domain knowledge that aims for a certain level of abstraction. However, the term 'model' is used differently in different contexts (Suppes, 1960). Therefore, in the following sections, we clarify its meaning in the context of this dissertation. We define (1) what a model (and a metamodel) is *conceptually* and (2) what a model is *technically* in the technological context of MDE, that is, in the modelware technological space. For the technical definition of a model, we look at modeling from an object-oriented perspective (Sec. 2.1.3). For a first understanding of what a model is conceptually, we discuss scientific modeling in general and modeling in software engineering in the next two subsections. However, for the final definition of a conceptual model we need concepts of language theory. Therefore this definition is presented in the section about SLE where we look at modeling from a language engineering perspective (Sec. 2.2.3).

### 2.1.1 Modeling in Science & Engineering

Creating models of things and phenomena has always been at the center of most scientific and engineering work. In his book on general model theory, Stachowiak (1973, pp. 131–133) characterizes a model by the following properties:

1. *Representation*: A model represents a (real or imaginary) *original* which can be a model itself.

2. *Abstraction*: A model does not capture all attributes of the original but only those relevant for a given *modeling purpose*.
3. *Pragmatism*: A model is created to represent an original only within the specific context of the modeling purpose, which means for *someone* at *some time*.

Often, a model is used instead of an original when performing a certain task with the original is difficult. Thus, a frequent purpose of using models is to repeatedly perform experiments which would otherwise be costly to perform with the original. For example, a small physical model of an airplane is tested in a wind channel, or a software model of a combustion engine is tested by means of computer simulation. Afterwards, the experiment can be performed once with the original to validate the results.

In order to decide what information about the original can be left out of the model, assumptions must be made about the context in which the model will be used. For example, information about the cabin interior of an airplane might be irrelevant for testing aerodynamics. If the assumptions are true, a correct model (i.e., correct with respect to the modeling purpose) can be used to predict the behaviour of the original. In other words, the model answers certain questions the same way as the original would. This is called *contextual substitutability* (Bézivin, 2005).

If a model can answer every possible question the same way as the original, then it is no model according to our understanding: it is then either the original (or a copy thereof) or a *definition*, i.e., a complete characterization. For instance, a function in mathematics is a binary relation (a set of pairs). It can often be completely characterized by an equational function definition provided with the function's domain and co-domain. Thus, a function definition is not a model of a function.

Another frequent purpose of modeling is *generalization*. Because a model does not contain all information of an original, one model can be a correct model of multiple originals. For instance, aerodynamics test results obtained from an airplane model without an actual cabin could be generalized for multiple actual airplanes with different cabin layouts. Similarly, a type in a programming language is a model. The type abstracts *over* different concrete data sets on which the same operations can be applied. We will use the expression 'abstract over', instead of 'abstract from', to indicate generalization.

Because a model is created for a specific modeling purpose, there are also multiple correct models for one original, for example, each model with a different purpose.

### 2.1.2 Modeling in Software Engineering & Model-Driven Engineering

In software engineering, we only deal with *conceptual models*. A conceptual model – from now on just 'model' – does not consist of physical objects but of concepts (in Sec. 2.2.3, with the help of language theory, we define the meaning of 'conceptual model' in this dissertation more precisely). One could argue that software engineering is concerned with creating models most of the time. A Java program, for instance, abstracts from the specifics of the machine it is executed on. However, in software engineering 'model' usually refers to a description with a higher level of abstraction than, for instance, a Java program.

MDE is a particular methodology in software engineering that, according to Schmidt (2006), can be characterized as follows:

- High-level models are the *primary artifacts*, i.e., models are the main things to be created, processed, and managed. 'Model-driven' emphasizes a contrast to earlier approaches to modeling in software engineering, where lower-level source code was the primary artifact and high-level models were only used for documentation etc.
- Models are *domain-specific*, i.e., they consist of concepts from the domain a software is built for, and not of concepts from the technology a software is implemented with. The goal is to effectively describe domain knowledge and solutions for problems within the domain using domain-specific concepts. Because there is no semantic gap, domain-specific models can be understood by *domain experts*.
- Tools for MDE support modeling by automatically checking models for *domain-specific constraints* which restrict how concepts can be combined in that domain (for instance, with the help of a *type system*). Thus, MDE tools help to create models which make sense in that domain and help to detect modeling mistakes early.
- *Model transformations* automatically transform high-level models to lower-level source code or to other models which may be equally abstract but serve a different modeling purpose. We discuss model transformations in more detail in Sec. 2.1.4.

Consequently, identifying the specific concepts and constraints of a domain is an important task in MDE – the result is called a *domain model*.

*Definition* 2.1 (domain model). A *domain model* $\mathcal{M}_D$ of a domain $D$ captures concepts, relations between those concepts, and constraints for combining those concepts, that are required for effectively describing knowledge and solutions for problems specific to $D$.

A domain model is a model of a domain. The modeling purpose is effective communication about knowledge and solutions specific to that domain. A *domain* (also: problem domain or application domain) can be a real-world domain like public transportation which includes concepts like busses and schedules, or a subdomain of software engineering such as graphical user interfaces (GUIs) which includes concepts like buttons and drop-down lists.

A domain model describes how models specific to that domain are structured. Therefore, creating domain models is called *metamodeling*. The prefix 'meta' (from greek: above, beyond) implies that a domain model says something about models created within that domain in general and thus stays at a 'higher' level (*meta-level*) than these models.[1] In MDE, a domain model is therefore called a *metamodel*.

### 2.1.3 Metamodeling: An Object-Oriented Perspective

Metamodeling in the modelware technological space has been heavily influenced by the *Object Management Group* (OMG), an international standards consortium focusing on

[1] The prefix 'meta' is used in this sense often. For example, metadata is data about data. A common application of metadata are schemas in database systems or structured document systems such as XML. The schema says something about how documents or database entries look like.

object-oriented technologies. The OMG's *Model-Driven Architecture* (MDA) is a specific approach to MDE and a set of related standards. MDA focuses on achieving platform-independent software development by generating executable source code from models which are preferably created with UML. Because of the MDA's code generation focus, models in MDA represent software.

Today, and particularly in this dissertation, MDE is interpreted less narrow than by MDA. Generation of executable source code and platform-independence are not always the ultimate goals, and UML plays a less important role.

However, another OMG standard which is part of MDA, the *Meta-Object Facility* (MOF), still plays a defining role in the modelware technological space. MOF enables metamodeling with object-oriented (meta-)concepts. One of these concepts is the *instance-of* relation which characterizes the relation between a class and an object which was created by instantiating that class. Because a class constitutes a type, the instance-of relation induces typing of instances. In fact, MOF was originally created as a type system for entities in OMG's *CORBA* standard. Based on the instance-of relation, MOF defines a *meta-layer hierarchy* where every element in one layer is an instance of a meta-concept in the layer above. Fig. 2.1 illustrates this hierarchy with an example of creating movie library software using the UML-centric MDA approach.

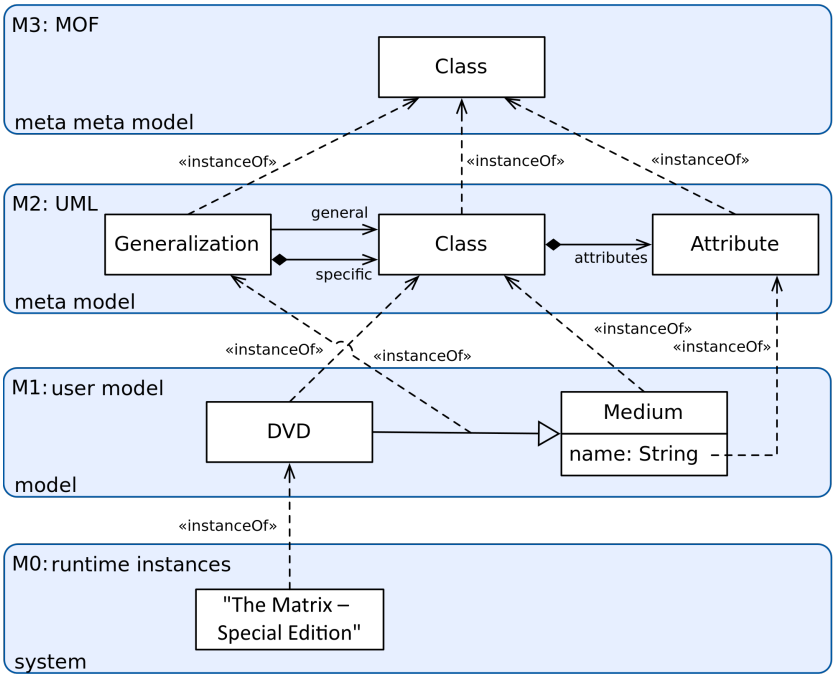



Figure 2.1: MOF meta-layer hierarchy (by Jens v. Pilgrim, based on OMG, 2004, p. 31)

Elements at the lowest meta-layer M0 are concrete runtime objects processed by the created software. These elements are instances of concepts defined in the M1 layer above, here for example, the concept of a DVD. Elements in M1 are again instances of concepts defined in the M2 layer above. The concept of a DVD, for example, is an instance of

the (meta-)concept Class. Importantly, not only the M1 concepts DVD and Medium are instances but also their relation – a DVD is a special kind of medium – is an instance of a meta-concept in the M2 layer above, here the concept of Generalization. Thus, elements in M2 clearly constitute a metamodel which describes what concepts can be used to create a model at M1 and how these concepts can be combined.

In this example, UML is used to describe the movie library model at M1. Therefore, at M2 concepts of UML are shown. They constitute the metamodel of models created with UML. The domain modeled by that metamodel is the domain of UML modeling. However, the concepts at M1 also clearly describe a domain; the domain of movie management. Thus, the UML model at M1 can be considered a metamodel which describes how models consisting of runtime objects at M0 are structured. Thus, the term 'metamodel' is relative. It is a *role* which is assigned depending on the layer you are looking from. The reason why here the set of M0 runtime objects is not called a model, is that from an MDA perspective the software is the original which a UML model represents. UML models are the ones to be created and managed by a software engineer using MDA.

As indicated in meta-layer M3 in Fig. 2.1, the MOF standard provides a metamodel consisting of very general concepts (like that of a class) which also UML is based on. Because the MOF metamodel is the metamodel of the metamodel of UML models, it is often called a *meta-metamodel*. This is also just a relative role, assigned because of the focus on UML models in MDA. However, there is something special about the MOF metamodel. The concepts in the MOF metamodel also need to be instances of some meta-concepts. Now, in order to avoid an infinite number of meta-layers, all elements in the MOF metamodel are instances of concepts in the MOF metamodel itself. The MOF metamodel is its own metamodel. MOF is so defining for the modelware technological space because most models in that space are indirectly based on the MOF metamodel, which means their metamodel (or meta-metamodel) is defined with concepts of the MOF metamodel.

The specific number of meta-layers depends on the particular MDE scenario. Four meta-layers are typical for MDA. However, when using EMF, for instance, only three meta-layers occur regularly. EMF is based on a simplified version of MOF called *Essential MOF* (EMOF) and provides an implementation of the EMOF metamodel called *Ecore*. In EMF, a metamodel describing the application domain (here at M1) is created directly from MOF concepts (here at M2), and this metamodel is represented by Java classes so that a model consists of Java runtime instances at M0. According to the MOF standard, any number of meta-layers greater or equal two is MOF-compliant.

We discuss the MOF meta-layer hierarchy and its focus on the instance-of relation more critically in Sec. 2.2.3. For now however, we have enough information to define what a model and a metamodel is *technically* in the modelware technological space. We make the qualification 'technical' because *conceptually* model and metamodel are relative roles. For precisely defining their conceptual meaning, we will look at metamodeling from a language perspective and show that it is helpful to think of metamodel as a model of a language, and of a model as an utterance of that language (Sec. 2.2.3).

However, there is a common structure how models are technically represented across the modelware technological space and it reflects the object-oriented MOF-interpretation

of metamodeling. Our modelware-specific definition of what a model (and a metamodel) is technically – which is adapted from Jouault and Bézivin (2006) – is based on two central concepts of object-orientation: typing by the instance-of relation and graphs (of objects).

*Definition* 2.2 (model (modelware)). A *modelware model* $m$ is a 3-tuple

$$m = \langle G, \mathcal{M}, \tau \rangle$$

where

- $G$ is a directed graph $G = \langle N_G, E_G, \gamma_G \rangle$ consisting of a finite set of nodes $N_G$, a finite set of edges $E_G$, and a function $\gamma_G : E_G \to N_G \times N_G$ which maps edges to their source and target nodes,
- $\mathcal{M}$ is a *modelware metamodel* $\mathcal{M} = \langle G_{\mathcal{M}}, ... \rangle$ with a directed graph $G_{\mathcal{M}} = \langle N_{G_{\mathcal{M}}}, ... \rangle$, and
- $\tau$ is a typing function $\tau : N_G \cup E_G \to N_{G_{\mathcal{M}}}$ which associates nodes and edges in $G$ (called $m$'s *model elements*) with nodes in $G_{\mathcal{M}}$, i.e., with their *meta-elements*, by an *instance-of* relation.

The above definition of a model relies on the following recursive definition of what a metamodel technically is, which conversely relies on the above definition of a model.

*Definition* 2.3 (metamodel (modelware)). A *modelware metamodel* $\mathcal{M}$ is a modelware model $\mathcal{M} = \langle G_{\mathcal{M}}, \mathcal{M}_{\mathcal{M}}, \tau, C \rangle$ which additionally contains a (possibly empty) finite set of *constraints* $C$, and whose (meta-)metamodel $\mathcal{M}_{\mathcal{M}}$ is either the MOF metamodel (or a similar one such as *KM3*[a]) or a modelware metamodel according to this definition.

A modelware metamodel $\mathcal{M}$ defines a (possibly infinite) set of modelware models $M = \{m \mid G_m \in \mathcal{P}(G_{\mathcal{M}}) \ \wedge \ \forall c \in C,\ c(m)\}$ where $\mathcal{P}(G_{\mathcal{M}})$ is the set of all graphs which can be constructed from instances of $\mathcal{M}$'s elements, and $c(m)$ denotes that a model $m$ satisfies a constraint $c$. We say that $M$ is the set of models which *conform to* $\mathcal{M}$.

[a] http://kermeta.org

The MOF metamodel itself is a metamodel according to this definition because it is its own metamodel. Physically, a model (and therefore also a metamodel) can come in different forms. For example at the runtime of a modeling tool, a model can be a graph of Java objects in a computer's main memory which are typed by the classes they are instances of. Alternatively, a model in its persistent form can be an XML document on a computer's hard drive together with its metamodel also stored as an XML document.

Importantly, following the common practice in the modelware technological space, we generally interpret a modelware model – not a conceptual model in general – as a *static structure* which, of course, can represent a dynamic system (Bézivin, 2005, p. 18).

### 2.1.4 Model Transformations

Besides (meta-)modeling – that is, creation of models – transforming those models is the key task in MDE. There are many different kinds of model transformation methods

and technologies. In this section, we define what a model transformation is and present selected categorizations of model transformations which are of particular relevance for this dissertation. Beyond that, we rely on terminology presented by Czarnecki and Helsen (2010) in their comprehensive taxonomy of model transformations.

### What is a Model Transformation?

We first have to distinguish between the actual process of transforming models and the description of that process. A model transformation description is a program[2] which processes models whereas – in its original meaning – a model transformation is the execution of this program by an *execution engine* given a particular set of models as input. We will, however, use the term 'model transformation' to refer to the transformation description and instead speak of the *execution of a model transformation* when referring to the process of transformation. This is closer to the typical use of the terms 'program' and 'program execution'. Furthermore, in its most general form, a transformation does not necessarily has to produce a model as an output.

*Definition* 2.4 (model transformation, model transformation description). A *model transformation* $\delta$ is a *program* which explicitly refers to a *source modelware metamodel* $\mathcal{S}$ defining a set of *source modelware models* $S$, and – when executed by an *execution engine* $\Gamma$ – accepts at least one model $s \in S$ as input.

Note that in this definition we refer to the technical modelware-specific definitions of model and metamodel, that we presented in the previous section. This is because the term 'model transformation' specifically refers to a transformation in the modelware technological space. Transformations in other technological spaces are, for instance, program transformation, term transformation, etc. As we did not define the output of executing a model transformation, we can categorize model transformations by their output.

*Definition* 2.5 (model-to-model transformation (M2M)). A *model-to-model transformation* $\delta$ is a model transformation which additionally refers to a *target modelware metamodel* $\mathcal{T}$ defining a set of *target modelware models* $T$, and – when executed – produces at least one element $t \in T$ as output. $\mathcal{T}$ can be the same as the source metamodel $\mathcal{S}$.

If source and target metamodel of a model-to-model transformation are the same, we speak of a *homogeneous* model transformation. If source and target metamodel are not the same, we speak of a *heterogeneous* model transformation. A model transformation tool should automatically check that the model taken as input and the model produced as output belong to the specified sets of valid input models and output models, respectively, e.g., by checking that the constraints of the source and target metamodel are satisfied. The latter is more challenging, especially when a *static* check is desired, that is, when it should be guaranteed before execution that a given model transformation produces a valid output. We will discuss this in more detail in Chap. 3 (Sec. 3.3.3) as *metamodel-awareness*.

[2]We will precisely define the term 'program' in Sec. 2.2.4

*Definition* 2.6 (model-to-text transformation (M2T), model-to-code transformation (M2C)). A *model-to-text transformation* $\delta$ is a model transformation which – when executed – produces a string of characters as output. If this string is supposed to be source-code in a given programming language, we can call the transformation more specifically a *model-to-code transformation* (M2C).

Czarnecki and Helsen (2010) argue that a model-to-text transformation is only a special case of a model-to-model transformation where the metamodel for the output is not explicitly defined. It is, however, defined implicitly because every meaningful transformation will have an output that conforms to some schema, language, etc. Nevertheless, in order to conform to some existing literature on the topic, Czarnecki and Helsen stick to the less specific interpretation where the output does not have to be a model. We follow this general definition. However, in this dissertation we are mainly concerned with model-to-model transformations. Therefore, we only distinguish between model-to-model and model-to-text transformations when the type is not clear from the context and otherwise refer to model-to-model transformations simply as model transformations. Fig. 2.2 shows the main concepts of a (model-to-)model transformation.

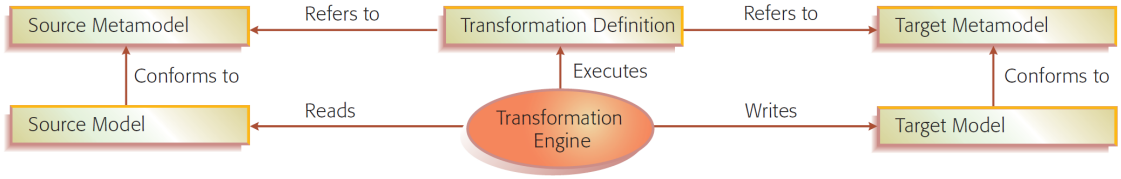



Figure 2.2: Main concepts of model transformations (from Czarnecki and Helsen, 2010)

### Bidirectional Model Transformations

Executing a model-to-model transformation with a source model as input returns a corresponding target model as output. Thus, a model transformation defines a *binary relation* between two sets of models. This idea of a model transformation is important when talking about *directionality* of model transformation, a categorisation that is particularly important in this dissertation. Model transformation can be unidirectional, bidirectional, or even multidirectional.

A *unidirectional model transformation* can only be executed in one direction. The roles of the sets defined by its source and target metamodel are fixed – the one serves as the set of possible inputs and the other as the set of possible outputs. A unidirectional model transformation is essentially a function whose domain and codomain are defined by source and target metamodel, i.e., they define the input and output 'type' of the transformation.

A *bidirectional model transformation* can be executed in two directions: either taking an element from the set of source models as input and returning an element from the set of target models as output, or vice versa. One could argue that with a bidirectional model transformation it makes no sense to speak of source and target because it depends on the direction of execution. However, it avoids misunderstandings to declare one set as the set of source models and one set as the set of target models.

A bidirectional model transformation is often implemented as a pair of two unidirectional model transformations, a forward transformation from source to target, and a backward transformation from target to source. Here it is particularly helpful to think of a model transformation as a definition of a relation. A pair of two unidirectional model transformations only comprises a valid bidirectional model transformation if both components of the pair correspond to the same relation, i.e., if they are *consistent* with another in the sense that they satisfy an invertibility property (Matsuda et al., 2007). Because it is difficult to guarantee this consistency of two unidirectional transformations there are special *bidirectional transformation languages* which allow a relation to be described in such a way that two inverse transformations can be automatically inferred so that they are consistent by construction. We will discuss this in more detail in Chap. 3 (Sec. 3.3.2).

Finally, there are *multidirectional model transformations* which define a relation between more than two sets. However, such relations can often be described with multiple bidirectional model transformations. Hence, we only cover bidirectional transformations.

## 2.2 Software Language Engineering

In early MDE it was considered that a few general-purpose modeling languages like UML would suffice for modeling all kind of systems. However, when describing models that are specific to a domain, it can be helpful to use a *domain-specific (modeling) language* (DSL) which also provides special notations for that domain (in contrast to the generic notations of a general-purpose language like UML). Therefore, nowadays, the development of new modeling languages is an important and frequent task in MDE. Software language engineering (SLE) is concerned with improving the process of creating software languages in general. In this section, we mainly look at SLE from the perspective of MDE with a focus on providing domain-specific language tooling.

In the following subsections, we present a set-theoretical definition of language, discuss how different language aspects can be described, and then look at metamodeling from a language perspective. Based on this, we define terms like ‘programming language’, ‘program’, ‘modeling language’, and ‘domain-specific language’. Afterwards, we define what an internal DSL is and discuss the internal DSL approach. We then discuss language tooling and define the terms ‘language workbench’ and ‘domain-specific workbench’.

### 2.2.1 What is a Language?

Originally, most software languages were textual and were described with the help of context-free grammars. In that context, ‘language’ often refers to a typically infinite set of strings and stems from the clearly defined term ‘formal language’ from language theory.

*Definition* 2.7 (formal language). A *formal language* $\mathcal{L}$ over an alphabet $\Sigma$ is a well-defined subset of the Kleene closure $\Sigma^*$ (the set of all possible strings over $\Sigma$). $\mathcal{L}$ can be generated by a grammar (Moll et al., 1988).

However, this notion of a language is not sufficient for MDE because of two reasons. First, because of the advent of graphical languages (like UML) a software language and

its (textual) representation cannot be seen as one and the same. Like in linguistics[3], we have to distinguish abstract *language utterances* and their concrete representations. For example, no matter if a sentence (a language utterance) is spoken or written, it is still the same sentence. Second, formal languages are only about structure of language utterances, i.e., their *syntax* (from greek 'sun taksis' = 'with arrangement'). They only describe whether a language utterance is syntactically correct but not whether it is semantically correct, i.e., whether it provides any useful meaning. Therefore, for defining software languages, Kleppe (2007) stays close to the definition of a formal language but abstracts from strings as the elements of a language and from a grammar to describe a language:

> "A *language* $L$ is the set of all linguistic *utterances* of $L$. (...) A *language description* of language $L$ is the set of rules according to which the linguistic utterances of $L$ are structured, optionally combined with a description of the intended meaning of the linguistic utterances."

However, if language utterances are independent from their representation and their meaning (and thus, by themselves have no meaning and no tangible form), a language has to be more than only the set of all its language utterances. It has to include what its utterances mean and how they can be represented. Therefore, in MDE (e.g., by Clark et al., 2008) a language is often divided into three *language aspects*: (1) *abstract syntax* (structure of abstract language utterances), (2) *concrete syntax* (concrete representation of language utterances), and (3) *semantics* (meaning of language utterances). Our set-theoretical definition of a language (which is adapted from Sadilek, 2011) reflects this division.

*Definition* 2.8 (language, software language). A *(software) language* $\mathcal{L}$ is a 3-tuple

$$\mathcal{L} = \langle A, \{C_1, ..., C_m\}, \{S_1, ..., S_n\} \rangle$$

where

- $A$ is the language's abstract syntax,
- $\{C_1, ..., C_m\}$ is the non-empty set of the language's $m$ concrete syntaxes, and
- $\{S_1, ..., S_n\}$ is the non-empty set of the language's $n$ semantics.

In this interpretation of a language the abstract syntax plays the central role. A language has only one abstract syntax and it determines the language's identity. Rather intuitively, a language can have more than one concrete syntax, e.g., a textual and graphical one. There has to be at least one concrete syntax. Less intuitively, and only because of practical reasons, we allow more than one semantics. Ideally, a language should have exactly one semantics because this way it can best fulfill its purpose: convey information unambiguously. We discuss each of these three language aspects in the following subsections.

[3] In theoretical linguistics, Chomsky (1965) already distinguished between the *surface structure* and the *deep structure* of a language: "It might be supposed that surface structure and deep structure will always be identical. (...) The central idea [...] is that they are, in general, distinct [...]".

### Abstract Syntax

We define the abstract syntax of a language as the set of all (syntactically correct) language utterances[4]. Thus, we stay close to the above definition of a formal language.

*Definition* 2.9 (abstract syntax). The *abstract syntax* $A$ of a language $\mathcal{L}$ is the set of all language utterances $u$ that are produced by an *abstract syntax description* $\Theta_A$.

In order to abstract from a grammar as a concrete means to produce the set of language utterances, we use the more general term 'abstract syntax description'. We define what an abstract syntax description consists of in the Sect. 2.2.2. Next, we define a language utterance as an independent entity which is an element of the abstract syntax.

*Definition* 2.10 (language utterance). A *language utterance* $u$ of a language $\mathcal{L}$ is an element of the abstract syntax $A$ of $\mathcal{L}$. It has an *internal structure* that conforms to the abstract syntax definition which produces $A$. Via a concrete syntax $C$ of $\mathcal{L}$ and a semantics $S$ of $\mathcal{L}$ a *representation* and a *meaning* is assigned to $u$.

### Semantics

The very purpose of a language is to communicate a meaning between two parties. Therefore, both parties have to share a *semantic domain*. A semantic domain is a set of meanings which is not tied to a specific language. The semantics of a language consists of a semantic domain and a mapping[5] which maps each language utterance to an element in the semantic domain. The fact, that a semantic domain is not tied to a language and that utterances are only mapped to meaning in that domain, can be illustrated by an example. Two people can talk in English or in German about a basketball game, e.g., referring to a foul, and could mean the exactly the same. The semantic domain of basketball neither belongs to the English language nor to the German language but exists independently.

*Definition* 2.11 (semantics). A *semantics* $S$ of a language $\mathcal{L} = \langle A, ... \rangle$ is a 3-tuple $S = \langle A, D_S, M_S \rangle$ consisting of $\mathcal{L}$'s abstract syntax $A$, a *semantic domain* $D_S$, and a *semantic mapping* $M_S : A \rightarrow D_S$ which is a total function that maps elements of the abstract syntax (i.e., language utterances of $\mathcal{L}$) to elements of $D_S$.

*Definition* 2.12 (meaning). A *meaning* $m$ of a language utterance $u \in A$ of a language $\mathcal{L} = \langle A, ... \rangle$ with a semantics $S = \langle A, D_S, M_S \rangle$ is an element of the semantic domain $D_S$ for which $M_S(u) = m$.

For software languages, there are *structure-only semantics* and *execution semantics*.

*Definition* 2.13 (execution semantics). An *execution semantics* is a semantics whose semantic domain is the *program domain* of a *programmable machine*, i.e., its semantic mapping maps language utterances to a valid set of instructions for that machine.

[4] There are different interpretations of the term 'abstract syntax' in the literature. Some refer to what we call internal structure of one language utterance, some to what we call the abstract syntax definition.

[5] With 'mapping' we do not necessarily mean 'function', i.e., we do not always imply an injective relation.

We use the term 'machine' as in Hopcroft (1979), e.g., a Turing machine is a machine in that sense. It is obvious that compiled programming languages like C++ have execution semantics. A C++ compiler maps a C++ program to a set of instructions that a specific silicon processor can execute. Importantly, execution semantics can be defined indirectly by mapping language utterances to utterances of another language which provides direct execution semantics. Structure-only semantics can be seen as a less powerful kind of semantics because most execution semantics also determine structure. For example, the structure-only semantics of UML class diagrams maps a class diagram to a set of object structures (instances of that class diagram). A comparable C++ program, however, can be compiled and executed and at the same time also determines the memory layout of objects.

#### Concrete Syntax

Similar to the semantics, we also define the concrete syntax of a language to consist of a *concrete syntax domain* and a relation that maps language utterances to elements of the concrete syntax domain, i.e., their representations (Clark et al., 2008).

In contrast to the semantic mapping which maps every utterance to a meaning, the relation between language utterances and their representations is not necessarily a function because there can be multiple representations of an utterance. For example, two strings which only differ in whitespace can represent the same Java program. Also, languages like Scala have optional concrete syntax elements, for example, parentheses can often be omitted, so that multiple textual representations match the same Scala program.

*Definition* 2.14 (concrete syntax). A *concrete syntax* $C$ of a language $\mathcal{L} = \langle A, ... \rangle$ is a 3-tuple $C = \langle A, D_C, M_C \rangle$ consisting of $\mathcal{L}$'s abstract syntax $A$, a *concrete syntax domain* $D_C$, and a *concrete syntax mapping* $M_C \subseteq A \times D_C$ which is a binary relation that relates elements of the abstract syntax $A$ (i.e., language utterances of $\mathcal{L}$) with elements of $D_C$.

*Definition* 2.15 (representation). A *representation* $r$ of a language utterance $u$ of a language $\mathcal{L}$ with a concrete syntax $C = \langle A, D_C, M_C \rangle$ is an element of the concrete syntax domain $D_C$ for which $\langle u, r \rangle \in M_C$.

### 2.2.2 Describing a Language

We make a distinction between a language and the description of that language. Analogously to our definition of a language, a language description consists of exactly one *abstract syntax description*, one or more *concrete syntax descriptions*, and one or more *semantics descriptions*.

*Definition* 2.16 (language description). The *language description* $\Theta_{\mathcal{L}}$ of a language $\mathcal{L} = \langle A, \{C_1, ..., C_m\}, \{S_1, ..., S_n\}\rangle$ is a 3-tuple

$$\Theta_{\mathcal{L}} = \langle \Theta_A, \{\Theta_{C_1}, ..., \Theta_{C_m}\}, \{\Theta_{S_1}, ..., \Theta_{S_n}\}\rangle$$

where

- $\Theta_A$ is an abstract syntax description,
- $\{\Theta_{C_1}, ..., \Theta_{C_m}\}$ is the non-empty, finite set of $m$ concrete syntax descriptions, and
- $\{\Theta_{S_1}, ..., \Theta_{S_n}\}$ is a non-empty, finite set of $n$ semantics descriptions.

Similarly as Clark et al. (2001), we define an abstract syntax description to consist of symbols, their relations (e.g., defined as production rules), and constraints.

*Definition* 2.17 (abstract syntax description). The *abstract syntax description* $\Theta_A$ of a language $\mathcal{L} = \langle A, ...\rangle$ with an abstract syntax $A$ is a 3-tuple

$$\Theta_A = \langle \Sigma, R, C\rangle$$

where

- $\Sigma$ is a non-empty, finite set of symbols,
- $R$ is a possibly empty, finite set of relations/production rules between symbols, and
- $C$ is a possibly empty, finite set of constraints.

A context-free grammar is an example of an abstract syntax description according to the above definition. Terminal rules determine the set of symbols, non-terminal rules determine the set of relations between these symbols, and there are no (i.e., an empty set of) additional context-based constraints.

Fig. 2.3 illustrates the structure of a language as we define it and how it is related to the general structure of a language description. Analogously to our definitions of the concrete syntax and the semantics of a language, their descriptions have to refer to a description of a concrete syntax domain and a semantics domain, respectively. Those domains are also sets (like the abstract syntax) and can also be described with set generating means. Furthermore, the mappings from language utterances to elements of those domains have to be described. There are different techniques for describing such mappings. We will not define the general structure of a concrete syntax definition and a semantics definition as we did with the abstract syntax definition. Instead, we will cover concrete modelware-specific techniques in the next section.

### Metamodel-Based Language Engineering

In MDE, a language's abstract syntax description is usually a modelware metamodel. The set of symbols is a set of standard data types and classes defined by MOF. Their relations are defined by custom classes and their relations, basically representing production rules for models. Constraints are described, for instance, in the *Object Constraint Language* (OCL), an OMG standard based on first-order predicate logic. It should be noted that in

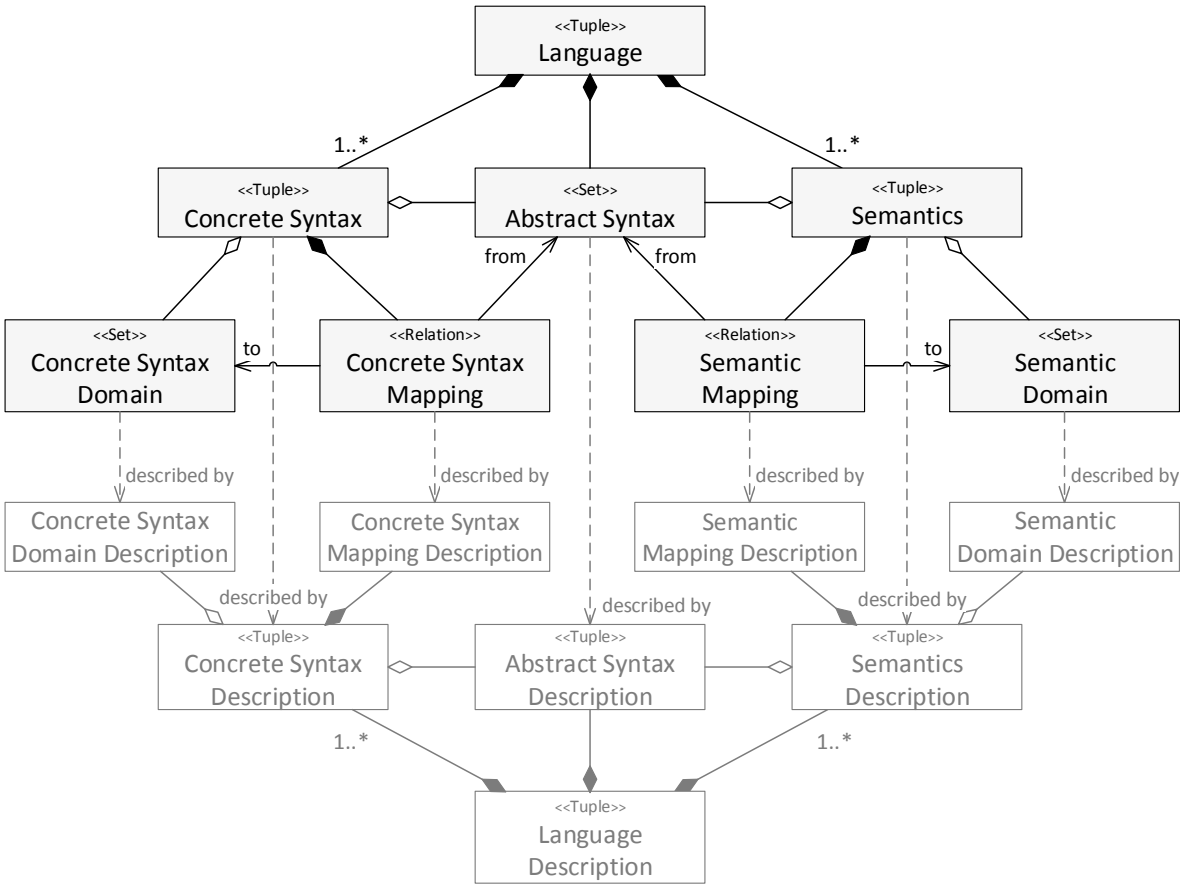



Figure 2.3: The different aspects of a language and their descriptions

MDE literature, often the term *static semantics* is used to describe the constraint-aspect of the abstract syntax description of a language. The term 'static semantics' is neither related to our notion of the semantics of a language nor to the term 'semantics' in natural language research and thus can be misleading. As a modelware metamodel defines a set of modelware models, language utterances of a *metamodel-based language* are models from that set.

*Definition* 2.18 (metamodel-based language). A *metamodel-based language* $\mathcal{L}_{\mathcal{M}} = \langle A, ..., ... \rangle$ is a language whose abstract syntax $A$ is described by a modelware metamodel $\mathcal{M}$, i.e., in $\mathcal{L}_{\mathcal{M}}$'s language description $\Theta_{\mathcal{L}_{\mathcal{M}}} = \langle \Theta_A, ..., ... \rangle$, $\Theta_A = \mathcal{M}$. A language utterance $u \in A$ is a modelware model which conforms to $\mathcal{M}$.

The semantics of a metamodel-based language can be described with different means. In MDE, generating executable code from models is often the goal. In this case, indirect execution semantics of a metamodel-based modeling language can be described by model transformations which transform utterances of that language (i.e., models created with that language) to code of an executable programming language like Java. This is often realized stepwise by first applying a set of model-to-model transformations – e.g., in order to augment models with platform-specific implementation details – and then

by applying a model-to-code transformation as an, ideally trivial, last step. Another transformation-based approach is to provide a transformation to a representation that conforms to a formal execution model like abstract state machines (Prinz et al., 2000). A non-transformation-based approach to describing execution semantics for a metamodel-based language is to provide an interpreter that directly executes language utterances of that language. The flexibility to provide code generators for different target platforms is one of the reasons why we allowed a language to have multiple semantics.

A textual concrete syntax of metamodel-based languages can be provided by describing a concrete syntax domain using a context-free grammar and by describing one-to-one mappings between elements of the metamodel and elements of the grammar. Alternatively, one can provide a bidirectional transformation between model and textual representation, i.e., a forward model-to-text transformation for transforming a model to its concrete textual representation, and a backward text-to-model transformation for creating a model from its textual representation. The former is often called *pretty printing*, the latter *parsing*. For a graphical concrete syntax a metamodel can be used to define the graphical concrete syntax domain. The two directions of the concrete syntax mapping for a graphical concrete syntax are also often called parsing and pretty printing. Fig. 2.4 shows how the three language aspects can be realized in a metamodel-based language.

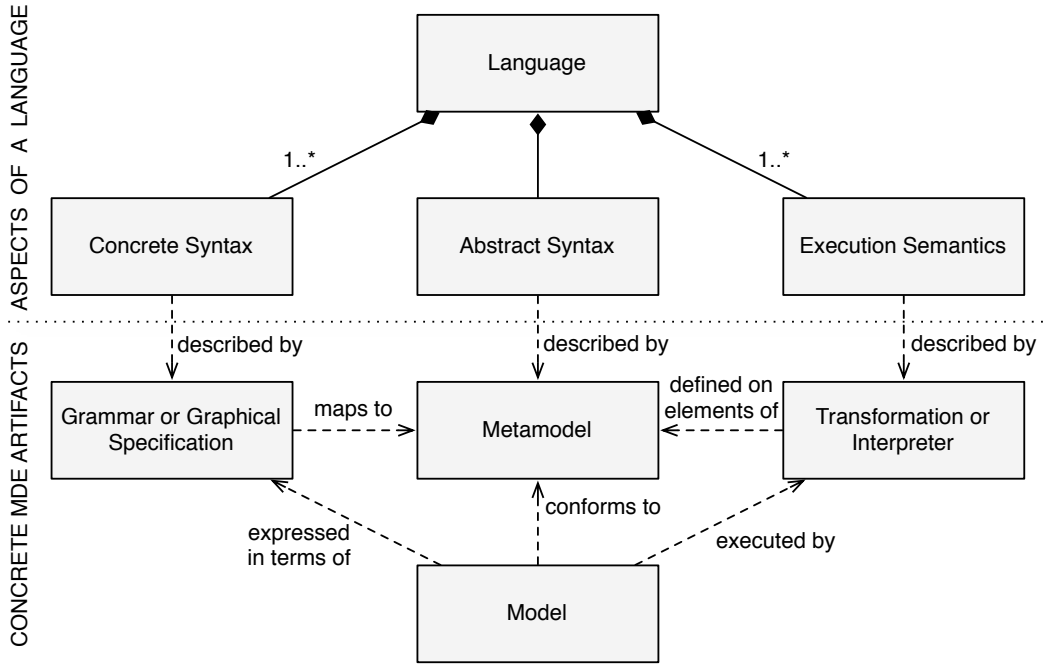



Figure 2.4: The aspects of a language and their realization in MDE

### 2.2.3 Metamodeling: A Language Engineering Perspective

In this section, we critically discuss the 'instance-of'-based meta-layer architecture from MDA. Then we use our understanding of what a language is (and how it can be described) to define the conceptual meaning of the terms 'model' and 'metamodel' in a way that is

both consistent with the technical, modelware-specific meaning we presented in Sec. 2.1.3 and with Stachowiak's general model theory we presented in Sec. 2.1.1.

The basic idea for this is that a model always *conforms* to the rules of a given language – its *modeling language*. Bézivin (2005) uses a geographical map as a typical example of a model. A map *represents* a part of the real world, it is greatly simplified (smaller, reduced detail, and two-dimensional), it is created for a given purpose (e.g., showing only routes accessible for cyclists), and it has a *legend*. The legend defines how we should read the map, i.e., to what (visual or textual) language it conforms. Often, parts of this language are implicit, e.g., north is often at the top of a map without explicitly noting it.

Reflecting this understanding of a model, Bézevin proposes to use 'conforms-to' and 'represented-by' as the two main relations in MDE instead of (over)using the 'instance-of' relation from object-oriented programming. In the original MDA standard the 'instance-of' relation is used to describe both of those relations which refer to fundamentally different modeling purposes. 'Conforms-to' refers to abstraction for generalization: a metamodel abstracts *over* the individual differences between models in a set. 'Represented-by' refers to abstraction for cost-effective substitution: a model of a system is easier to statically check than the system itself.

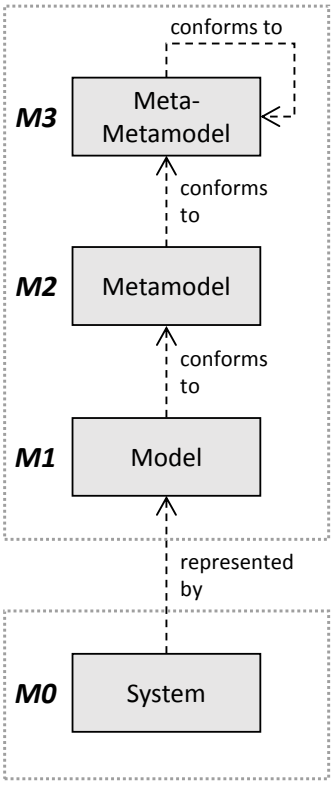



Figure 2.5: 3+1 meta-layers

Bézivin argues that the intuitive understanding of "A is a model of B" is closer to "B is represented by A" than to "B conforms to A"[6]. Therefore, the often found statement "a metamodel is a model of a model" is not wrong but can be misleading. Fig. 2.5 shows a version of the four-layer meta-layer architecture we showed in Sec. 2.1.3 using the new relations: A model represents something (its original) and conforms to its metamodel. This metamodel conforms to a meta-metamodel which conforms to itself.

Though both types of relations can also be described by 'instance-of' (or the other way round by 'defines a set of'), it can be illustrated how different they are by swapping them. It makes little sense to say that an original conforms to its model and it is not intuitive to say that a metamodel is a representation of one of its models. But what does a metamodel represent (i.e., what is it a model of)? As we showed in the previous sections, in MDE, a metamodel describes the abstract syntax of a language, but a language is more than its abstract syntax. Thus, a metamodel is a model of a language, that abstracts from the language's concrete syntax and semantics. For illustrating these relations between model, metamodel, and language, Génova (2005) proposed to arrange the different meta-layers as stairs instead of layers (Fig. 2.6.)

[6] "B conforms to A" could be stated as "A is a *role-model for* B" bringing the term 'model' closer to the template notion of a class.

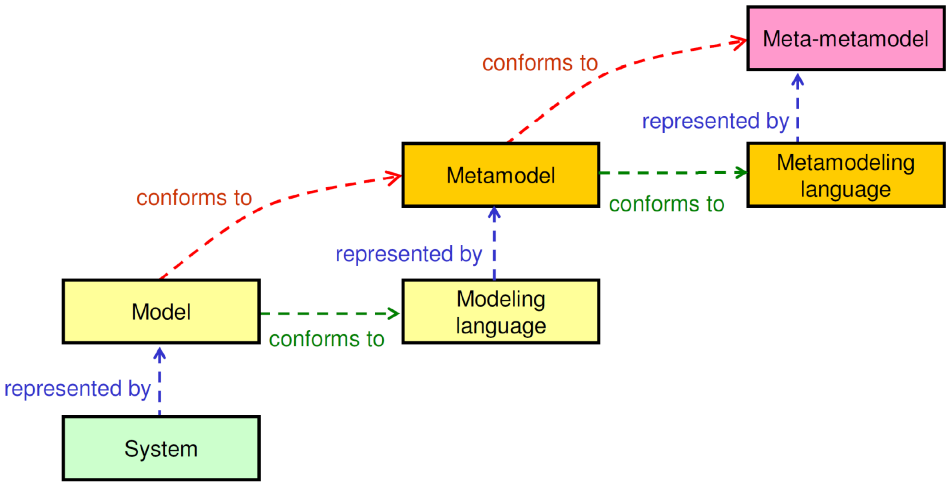



Figure 2.6: Meta-layers arranged as stairs instead of a stack (from Génova, 2005)

Note that because in MDE the metamodel describing the abstract syntax is the central part of a language, a model is said to both conform to its metamodel and to its modeling language. Thus, the statement "model $m$ conforms to a metamodel-based language $\mathcal{L}_{\mathcal{M}}$ with a metamodel $\mathcal{M}$" implies "model $m$ is an element of the set of models defined by $\mathcal{M}$". Thus, a model is a language utterance of the modeling language it conforms to.

*Definition* 2.19 (model (conceptual)). A *conceptual model* is an *abstract representation* of an original. It abstracts from details irrelevant for a given *modeling purpose* and it *conforms to* a given *modeling language*, i.e., it is a *language utterance* of that language. Technically, a model can be implemented, for example, as a modelware model.

*Definition* 2.20 (metamodel (conceptual)). A *conceptual metamodel* is a model of a language. It describes the abstract syntax of this language. Technically, a metamodel can be implemented, for example, as a modelware metamodel.

The above definitions describe our general understanding of a model and a metamodel independently from the concrete technical implementation. However, in the context of this dissertation all models and metamodels are implemented technically as modelware models. Therefore, from now on, we will not always distinguish between the general and the technical notion. Only if we want to stress one specific notion, we will use the specific terms 'modelware model' (technical), 'conceptual model'[7] (non-technical), or 'language utterance' (language-related) instead of just using 'model' or 'metamodel', respectively.

### 2.2.4 Modeling Languages, Programming Languages, and DSLs

When speaking about software languages, it is often distinguished between different kinds of languages, e.g., programming languages, modeling languages, and DSLs. In the following subsections, we define those terms using the previously defined language concepts.

[7] In some computer science literature, the term 'conceptual model' is specifically used for what we defined as a domain model. For us, a domain model is a specific kind of a conceptual model.

### Modeling Languages & Programming Languages

Often, a distinction is made between *modeling languages* and *programming languages*. In many cases this distinction is based on the perceived level of abstraction. Modeling languages such as UML are often described as providing a higher level of abstraction than typical programming languages such as C++ which let you express fine-grained implementation details. However, as this is rather subjective and also gradual (e.g., Java abstracts more from the hardware than C++), we will not distinguish between modeling languages and programming languages based on the level of abstraction. Every language utterance abstracts from the real world by using simplified concepts defined by the language. Consequently, we have defined that every language utterance is a model. For us, *every software language is a modeling language* because it is used to create models.

However, there is another way to distinguish between a modeling and a programming language. Programming languages are usually executable whereas modeling languages are not necessarily executable. Modeling languages are often described more generally as languages that allow expressing structured data in a well-defined way. We follow this distinction and align it with our distinction between execution semantics and structure-only semantics. Thus, we define the term 'programming language' to be synonymous with 'executable language', i.e., a language which provides execution semantics:

*Definition* 2.21 (programming language, executable language). A *programming language* is an *executable language*. An executable language is a language with execution semantics.

As mentioned before, execution semantics also often define structure. Thus, programming languages are also often modeling languages (e.g., we can use Java to describe an object structure) but modeling languages are often no programming languages. One can, however, define execution semantics for any modeling language to make it a programming language. Based on the definition of programming language we define what a program is.

*Definition* 2.22 (program). A *program* is an utterance of a programming language.

This is consistent with the traditional definition of a program to be a set of instructions. The elements of the internal structure of a programming language utterance can be transformed into or serve themselves as instructions for a machine. Consequently, every program is a conceptual model (a model of the machine yielded by passing the program to a programmable machine) but not every model is a program that can be executed.

### Domain-Specific Languages & General-Purpose Languages

Another common distinction among software languages is that between *general-purpose languages* (GPLs) and *domain-specific languages* (DSLs, formerly also referred to as *little languages*, Bentley, 1986). The general understanding of this distinction is that a DSL is a language which is tailored to a narrow application domain, so the variety of problems it can be applied to is rather limited, whereas a GPL has a wider application domain which potentially subsumes several more specific domains (Spinellis, 2001; Landin, 1966). Thus, a GPL is more versatile. However, this again is a gradual distinction. A language can

be more or less domain-specific. In a certain sense, every language has a limited domain and thus is domain-specific. Ideally, we would be able to compare languages by the size of their application domain. However, there is no agreement on how to measure the size of an application domain. One could say that a domain which contains all elements of another, plus more elements, is bigger (i.e., there is a subset relation). However, such a subset relation cannot be defined for disjunct domains. Therefore, we define what a DSL is in terms of its goals.

The main goal of a DSL is to be particularly *expressive* in its domain, in the sense that more can be said with less. This is possible because with a narrow application domain, one can agree on more implicit assumptions and less has to be stated explicitly. This way, domain concepts can be expressed concisely. Thus, a DSL can achieve a higher level of abstraction within its domain than a GPL at the cost of being less versatile. The goal of this abstraction is efficient communication of domain knowledge. Like a model, a DSL tailored to one domain should not be used to describe knowledge in another domain because made assumptions may not be true. This is known as the *modeling gap*.

However, abstraction is not the only goal of a DSL and there may be DSLs which do not provide more abstraction than a GPL at all. The other goal of a DSL is the limitation itself which can help to prevent users of the DSL to create models that do not make sense in the particular domain. Also, the limitation allows language tooling to provide better user assistance, as the assistance can be better tailored for the domain. As there is no general agreement in the literature on what a DSL is exactly[8], we define it by its goals.

*Definition* 2.23 (domain-specific language). A *domain-specific language* (DSL) is a language with a narrow application domain. Its goals are to be particularly expressive in its domain, and to prevent users from creating models which make no sense in that domain.

*Definition* 2.24 (domain-specific model). A *domain-specific model* is a language utterance of a domain-specific language.

A good example of a DSL according to the above definition is SQL. Database queries can be expressed much more concisely than, say, a general-purpose programming language like Java. Furthermore, it is hard to express anything else in SQL than database queries. Implementing an application with a graphical user interface (which is a different application domain) is hardly possible using SQL because one cannot describe a constantly running user input loop. Take on the other hand an assembly language for an x86 processor. It is a general-purpose language whose application domain is only limited by the processor architecture. One can implement a wide range of applications, including graphical user interfaces and database queries. However, describing a database query would need lots of assembly code. Thus, it is not very expressive in that domain.

Combining the distinction between GPLs and DSLs with the distinction between programming and modeling languages, one could speak of general-purpose programming languages, general-purpose modeling languages, domain-specific programming languages,

[8] e.g., see M. Fowler's discussion about a general definition: http://martinfowler.com/bliki/DslBoundary.html

and domain-specific modeling languages. Java, Scala, C++, etc. are examples of general-purpose programming languages because they are versatile and executable. SQL, being also executable, could be considered a domain-specific programming language. UML is an example for general-purpose modeling language, as it is very versatile but not executable. Such a modeling language can be provided with execution semantics, as shown by *Executable UML*, which makes it a programming language according to our definition. HTML could be considered an example of a domain-specific modeling language, because it has structure-only semantics and is specifically tailored for describing web pages.

### 2.2.5 Internal and External Domain-Specific Languages

The classifications presented in the previous section are about properties of a language, for example, the size of its domain or whether it has executable semantics. Independently from its properties, languages can also be classified by the way they are created, that is, how they are described and how the language tools are implemented. There are two substantially different approaches for this. One way is to create a language independently, by describing all of its aspects and implementing new tooling specifically for this language, for example, a compiler and an editor[9]. Languages implemented this way are called *external languages* or *independent languages*.

The other way is to embed a language into an existing language – called the *host language* – and to reuse the host language's tooling. Languages implemented this way are called *internal languages* or *embedded languages* (Mernik et al., 2005). Because the application domain of an internal language is usually smaller than the domain of its host language which is often a GPL, internal languages are mostly referred to as *internal DSLs*. In order to conform to the usage of the term in the literature and because, as mentioned before, a language is in a certain sense always domain-specific, we will use only the term 'internal DSL' from here on, referring to any language implemented by embedding, no matter if it is particularly domain-specific.

In this dissertation, we apply both of these approaches to software language development. However, because the external approach is the more traditional, well-know approach – studied, for instance, for decades in compiler construction – we will focus on describing the internal DSL approach. In the following four subsections, we define what an internal DSL is, how it is described, and compare it with the external approach.

#### What is an Internal DSL?

The easiest (but not very precise) way to describe an internal DSL is to say that it is a software library (written in its host language) whose usage *feels* like using an external DSL. An example of an internal Java DSL (i.e., one that uses Java as the host language) is *jOOQ*[10] which embeds SQL-like queries into Java as shown in the following listing.

[9] This does not mean that the language tools need to be implemented manually. They could be generated from the language description or be realized as parameterized versions of generic language tools. We will discuss this in more detail in Sec. 2.2.6

[10] *Java Object Oriented Querying* (http://jooq.org)

```
create.selectFrom(table("AUTHOR").as("a"))
  .where(exists(selectOne()
    .from(table("BOOK"))
    .where(field("BOOK.STATUS").equal(field("BOOK_STATUS.SOLD_OUT")))
    .and(field("BOOK.AUTHOR_ID").equal(field("AUTHOR.ID")))));
```

The above is valid Java code that looks a bit like an SQL statement. This is mostly achieved by the *fluent interface* pattern (explained in more detail in Sec. 2.4.3) which allows method calls to be chained without having to refer to the same object repeatedly. However, the above code still contains a lot of *syntactic noise* like dots and parentheses for method invocations. Scala, in contrast, allows omitting dots and parentheses in many situations and supports operator overloading. We say that Scala has a more *flexible* concrete syntax than Java. Therefore, the following statement described with the same DSL embedded into Scala looks more like an SQL statement. It also consists mostly of method calls but because of Scala's flexible concrete syntax, it is easier to achieve the feel of an external DSL.

```
create
  select ( AUTHOR as "a" )
  from BOOK
  where ( BOOK.STATUS === BOOK_STATUS.SOLD_OUT )
  and ( BOOK.AUTHOR_ID === AUTHOR.ID )
```

In certain sense, an internal DSL *extends* its host language by adding pre-defined, particularly expressive, domain-specific constructs. However, if using the internal DSL is interpreted as using only those concepts defined by the internal DSL and those parts of the host language which are meant to be reused in the internal DSL, it is better to think of an internal DSL as a *customized subset* of its host language. Because we defined a language as a 3-tuple, and not as a set, we have to clarify this 'subset relation' between an internal DSL and its host language.

Clearly, the abstract syntax of an internal DSL is a subset of the abstract syntax of its host language because every utterance of the internal DSL is also an utterance of the host language. Furthermore, an internal DSL completely inherits from the host language how language utterances are mapped to their representation and their meaning, respectively. For example, a program written in an internal Java DSL is compiled to Java bytecode the same way as any other Java program. Because the domain[11] of these mappings – the abstract syntax – is smaller, so is their image within the concrete syntax domain and the semantic domain, respectively. Thus, also these relations are subsets of the corresponding relations of the host language. We could say that a language, as we defined it, is a subset of another language, if all three sets which the language consists of (i.e., abstract syntax, concrete syntax mapping, and semantic mapping; the concrete syntax domain and the semantic domain are only referenced) are subsets of the corresponding sets of the host language. Therefore, tools created for working with the bigger sets of the host language also work with the smaller subsets of the internal DSL without any modification.

[11] in the mathematical sense of 'domain' as the set of valid input elements of a function

*Definition* 2.25 (internal DSL). An *internal domain-specific language* $\mathcal{I}$ is a *customized subset* of an existing language $\mathcal{H}$ (the *host language*), in the sense that the abstract syntax of $\mathcal{I}$ as well as all of $\mathcal{I}$'s concrete syntax mappings and semantic mappings are subsets of the corresponding sets of $\mathcal{H}$. Hence, language tools for $\mathcal{H}$ can be used without modification for creating, displaying, and processing language utterances of $\mathcal{I}$. By providing custom abstractions, $\mathcal{I}$ achieves to be more expressive in $\mathcal{I}$'s specific application domain than $\mathcal{H}$ without these abstractions.

### Describing an Internal DSL

The concrete syntax description and the semantics description of an internal DSL are identical to those of the host language. Therefore, for describing an internal DSL, only the tailoring of the host language's abstract syntax has to be described. This tailoring is usually described only productively (not by defining additional constraints), e.g., by defining custom classes using concepts from the host language. Thus, the host language is the meta-language for describing the abstract syntax of its internal DSLs.

*Definition* 2.26 (internal DSL description). A description $\Theta_{\mathcal{I}}$ of an internal DSL $\mathcal{I} = \langle A_{\mathcal{I}}, ..., ... \rangle$ with a host language $\mathcal{H} = \langle A_{\mathcal{H}}, ..., ... \rangle$ is a tuple $\Theta_{\mathcal{I}} = \langle \Theta_{\mathcal{H}}, \Theta_{A_{\mathcal{I}}} \rangle$ where $\Theta_{\mathcal{H}} = \langle \Theta_{A_{\mathcal{H}}}, ..., ... \rangle$ is a description of $\mathcal{H}$, and $\Theta_{A_{\mathcal{I}}}$ is an abstract syntax description of $\mathcal{I}$. $\Theta_{A_{\mathcal{I}}}$ is a model that conforms to $\mathcal{H}$, i.e., $\Theta_{A_{\mathcal{I}}}$ is an element of $\mathcal{H}$'s abstract syntax $A_{\mathcal{H}}$.

Fig. 2.7 illustrates how an internal DSL consists of subsets of the corresponding sets of the host language, and that the abstract syntax description of an internal DSL itself is an utterance of the host language.

Following our definitions, any traditional software library can be considered an internal DSL within the programming language it is implemented in. The set of programs which can be written solely by using functions and classes defined in the library, is a subset of all the programs which can be written with the library's host language. In this case the functions and classes defined in the library are the production rules and the symbols that comprise the abstract syntax description.

Another example for creating internal DSLs is the profile mechanism of UML. Notably, in addition to productive means to describe custom elements (e.g., stereotypes), UML also allows the abstract syntax of the internal DSL to be restrictively tailored by specifying DSL-specific constraints.

### Tailoring Concrete Syntax and Semantics of an Internal DSL

Although formally an internal DSL is nothing else than a traditional software library, the term only came up recently. It refers more specifically to a library which heavily utilizes the flexible concrete syntax of its host language to achieve the look-and-feel of an external DSL. Thus, languages which provide a flexible concrete syntax and powerful means for creating custom abstractions are considered particularly well suited as host languages for internal DSLs. We briefly illustrated this above by comparing the embedding of a query DSL into Java and into Scala. Dynamically typed languages like Ruby or Lisp are often

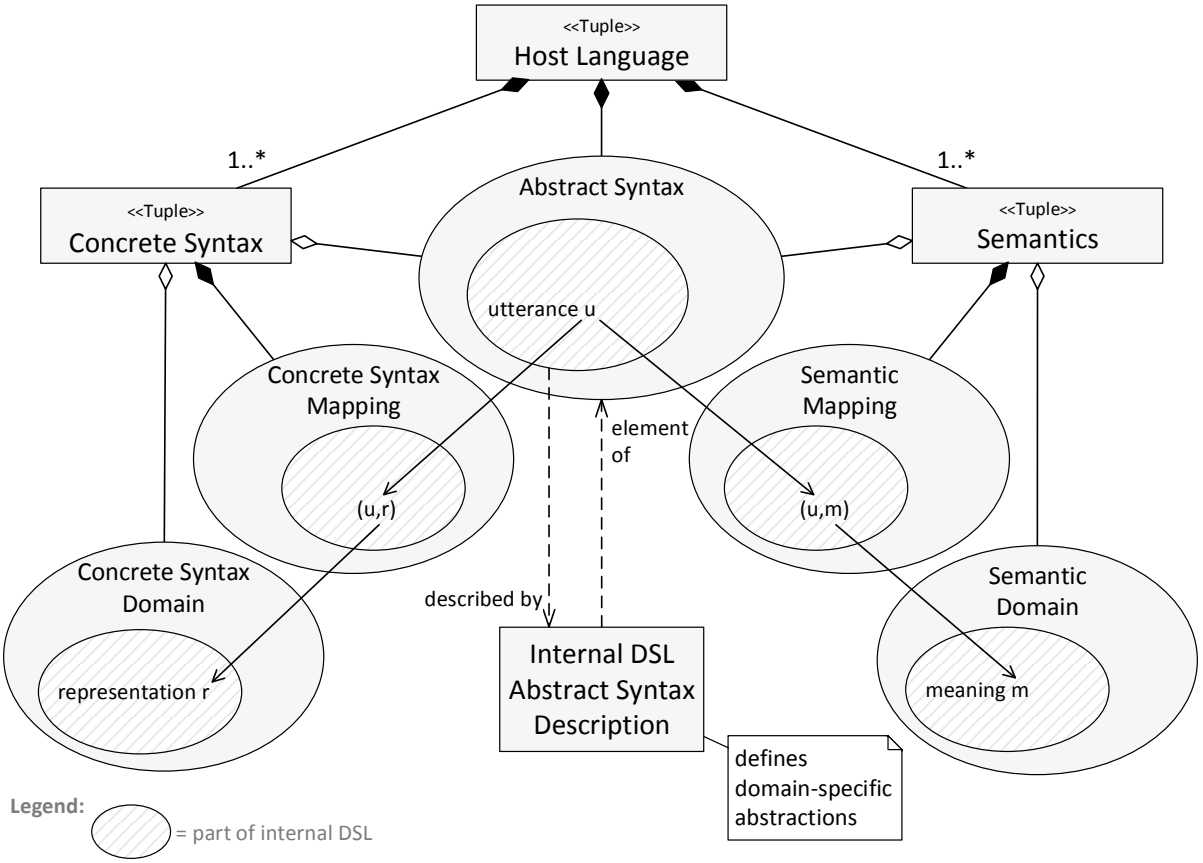



Figure 2.7: An internal DSL consists of subsets of its host language's sets

considered better suited than statically typed languages like Java because it is easier to achieve a domain-specific syntax without having to integrate type annotations (Günther and Cleenewerck, 2010). In Chap. 4 (Sec. 4.1.3), we will compare potential host languages for the internal transformation languages which we develop in this dissertation, and conclude that Scala meets our specific requirements best.

Nevertheless, creating a domain-specific concrete syntax (and semantics) just by using concepts of the host language is often challenging. Internal DSLs often involve elaborate conversions of the actual domain-specific concepts (i.e., elements of the internal DSL's abstract syntax description) to conceptually unrelated concepts only to be represented by the host language's concrete syntax in the desired way. If actual domain concepts are clearly separated from concepts just needed for concrete representation, the internal DSL description will contain an explicit domain model (as recommended by Fowler, 2010). One could then argue that the description of an internal DSL also includes a description of an indirect concrete syntax. Analogously to indirect transformation-based execution semantics, utterances of the internal DSL are transformed to utterances of the host language which are then mapped to the concrete syntax domain of the host language.

#### Advantages and Disadvantages of the Internal DSL Approach

In Chap. 4 (Sec. 4.1.2), we will explain our decision to develop the transformation languages, which are presented in this dissertation, as internal DSLs, based on our specific requirements. Here, we only briefly sketch the general advantages and disadvantages of the internal DSL approach in comparison to the external DSL approach.

The main advantage of the internal DSL approach is that no effort has to be put into the development of the DSL tooling and its maintenance because the host language's tooling can be reused (Fowler, 2010). Of course, this advantage depends on how good existing tooling for the host language is. Apart from tool reuse, also reusing parts of the host language itself can prove to be an important advantage. For instance, many DSLs need to provide simple arithmetic expressions which – with an internal DSL – simply can be reused from the host language. Furthermore, an internal DSL can be seamlessly integrated with other libraries of the host language, including other internal DSLs.

It can be considered both an advantage and a disadvantage that an internal DSL can usually be mixed freely with its host language. Especially programmers appreciate this possibility to break the boundaries of the internal DSL, as it provides versatility when needed. However the possibility to break the boundaries of an internal DSL stands in direct contrast to the second goal of a DSL, namely to prevent a user from creating models that make no sense in the DSL's domain. Because an internal DSL reuses the tooling of its host language, the tooling can usually not enforce that a user only uses concepts of the internal DSL.

The main disadvantage of an internal DSL is that both the reused tooling and the concrete syntax can often not be as domain-specifically tailored as the specifically developed custom tooling and the freely designed concrete syntax of an external DSL. This is especially an issue, if the DSL is meant to be used by domain-experts who are accustomed to using special graphical notations unique to their domain (i.e., not boxes and arrows). Furthermore, it can sometimes be hard to separate the descriptions of the different language aspects of an internal DSL as clearly as with an external DSL. This can result in a language implementation that is hard to understand and to maintain.

Taking these advantages and disadvantages into consideration, the internal DSL approach is often favorable when the problem domain is close to the domain of the host language, e.g., if it is a DSL mainly to be used by programmers familiar with the host language. The external DSL approach is often favorable when a completely tailored concrete syntax is important because the DSL is to be used by highly specialized domain-experts.

### 2.2.6 Creating Domain-Specific Language Tooling

Today's IDEs for popular general-purpose programming languages like Java integrate a multitude of tools which make working with the language more comfortable and assist the language user in all sorts of ways. Many DSLs, however, have only a small user base in comparison to GPLs like Java or UML. Therefore it is hard to justify high costs for developing rich-featured tooling for a DSL. The internal DSL approach presented in the previous section is one approach to cost-effective tool development for DSLs. However,

as we have seen, this approach has disadvantages especially when a DSL is meant to be used mainly by highly specialized domain experts who may not have any experience with general-purpose programming languages. In such cases it is still required to create specially tailored DSL tooling, in other words, to create an external DSL (or several). Such specially developed, rich-featured tooling which integrates tools for one or several DSLs which belong to the same domain is called a *domain-specific workbench*.

*Definition* 2.27 (domain-specific workbench). A *domain-specific workbench* for a domain $\mathcal{D}$ with a non-empty set of languages $L_{\mathcal{D}} = \{\mathcal{L}_1, ..., \mathcal{L}_n\}$ which are tailored for (subdomains of) $\mathcal{D}$, is a set of integrated tools which make the creation, management, and processing of language utterances of $\mathcal{L}_1, ..., \mathcal{L}_n$ comfortable.

There are several domain-specific workbenches which have been developed manually. *Bioclipse*, for example, is a workbench for the domain of bioinformatics that integrates tooling for multiple graphical and textual DSLs including editable domain-specific views (Spjuth et al., 2007). Fig. 2.8 shows *Bioclipse*'s editor for a graphical molecule design DSL. In this dissertation, however, we are only concerned with (at least partly) automated domain-specific workbench development.

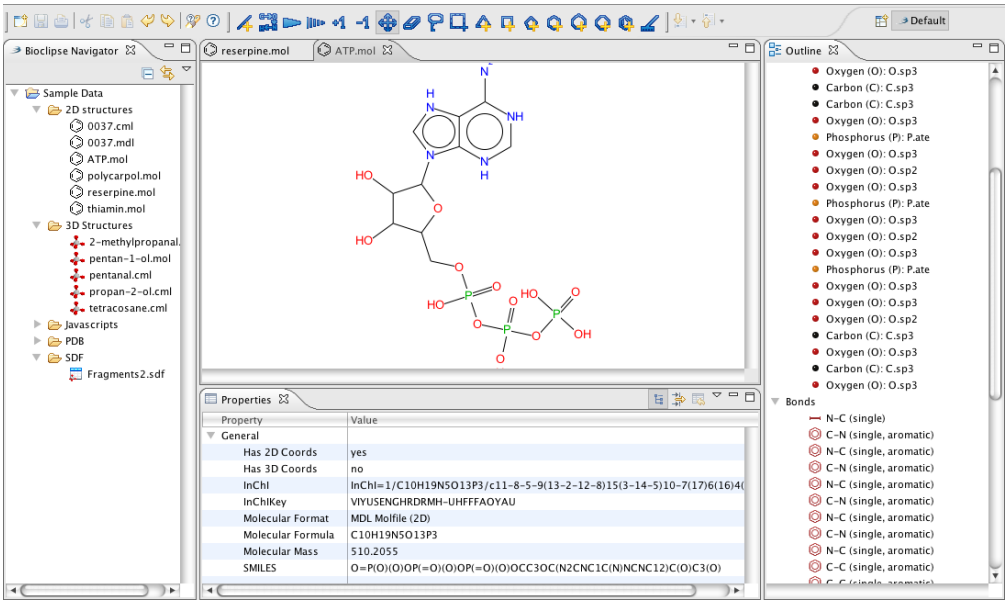



Figure 2.8: The *Bioclipse* domain-specific workbench for bioinformatics

### Automated Development of Language Tooling

The idea of automated development of language tooling is to automatically create language tools like editors, compilers, etc. based on the description of (the different aspects of) a language. In general, there are two approaches to this: *parameterization* of generic language tooling and *generation* of (source code of) language-specific tooling.

With the former approach, generic language tooling gets parameterized with a (possibly partial) description of the language, in order to provide special support for the described language. A typical example of this approach is the parameterization of text editors for custom syntax highlighting. This is of course a very simple example of parameterization and the model of the language here is a very limited one. A more advanced version of the parameterization approach is *projectional editing* (Völter and Solomatov, 2010). Here, edits made to a domain-specific editor are immediately projected to an abstract underlying representation of a model. Thus, with a projectional editor, one directly modifies an element of the abstract syntax which is then projected in the other direction to display the updated concrete representation. In the projectional editing approach, a metamodel and its projections are described, and a generic language tooling is parameterized with them. Tools which implement this approach are, for instance, the *Meta Programming System* (MPS) and the *Intentional Domain Workbench*[12]. Although it is a very promising approach, projectional editing is not in the scope of this dissertation. Instead, our work focuses on the generation approach.

With the generation approach, the complete source code of a language-specific tool such as an editor gets generated from models which describe the different language aspects. The Eclipse Modeling Framework (EMF, discussed in the next section) in general and several technologies based on EMF such as *GMF* and *Xtext* are examples of this approach (Seehusen and Stølen, 2011). It is essentially an application of MDE to the development of tooling for (modeling) languages. We will discuss this approach in more detail in the next chapter.

For either of the two approaches to automated language tool development, the different language aspects have to be described using appropriate meta-languages. For creating and processing language utterances of such meta-languages, (meta-)language tooling is needed (Fischer et al., 2005). A tool which integrates several meta-language tools for describing different aspects of a language and allows rich-featured tooling for the described language to be automatically creating (using either of the two approaches) is called a *language workbench*[13] (Tolvanen and Kelly, 2009; Kats and Visser, 2010; Erdweg et al., 2013).

*Definition* 2.28 (language workbench). A *language workbench* is a tool which integrates several meta-language tools for describing different aspects of a language using appropriate meta-languages and – based on these descriptions – allows the automated creation of a domain-specific workbench for the described language.

Thus, a language workbench is a general environment for describing languages and creating language tooling. A domain-specific workbench in contrast is a specific set tooling for one particular domain, and can be created using a language workbench. Because of the meta-level focus of a language workbench, one could say – using the instance-of relation – that a domain-specific workbench is one particular domain-specific *instance* of a language workbench.

[12] see http://jetbrains.com/mps and http://intentsoft.com, respectively

[13] The term 'language workbench' was coined by M. Fowler's in a blog post called "Language Workbenches: The Killer Application for DSLs" that was later edited into a book on DSLs (Fowler, 2010).

## 2.3 The Eclipse Modeling Framework

According to the project's website[14], EMF is *"a modeling framework and code generation facility for building tools and other applications based on a structured data model."* EMF is based on *Eclipse* which, originally, was only a Java IDE (written in Java), then evolved into an extensible and customizable IDE for different programming languages, and finally evolved into a generic platform for application and tool development (albeit keeping a focus on software language tooling). With EMF, one can generate *Eclipse*-based modeling tools based on a metamodel. EMF provides three alternatives for describing a metamodel: as a UML class diagram, as an XML schema, or as annotated Java code. Thus, *"EMF unifies [...] Java, XML, and UML."* (Steinberg et al., 2008, p. 30). From a metamodel which is described using any of these options, EMF generates Java classes whose instances (at runtime) make up an EMF-based model which conforms to the metamodel. EMF provides a Java API for navigating and modifying such a model, and generates two basic metamodel-specific editors for model creation and modification: a tree editor and a (class) diagram editor. Furthermore, EMF provides persistence functionality for saving and loading of models and metamodels, based on the *XML Metadata Interchange*[15] format (XMI). Most importantly, EMF enables interoperability of (language) tools which have been built using EMF. This way, EMF constitutes a *tool ecosystem* which is (currently) omnipresent in the modelware technological space.

### 2.3.1 The Ecore Meta-Metamodel and Single Containment

At the heart of EMF is the *Ecore* (meta-)metamodel. It is the metamodel which EMF-based metamodels have to conform to. According to EMF's project lead, Ecore is the defacto reference implementation of the *Essential MOF* standard (EMOF)[16], which is a cut down version of the *Complete MOF* standard (CMOF). Thus, Ecore uses a subset of UML's class diagram concrete syntax (e.g., there is no concept of 'association'). Like the (C)MOF (meta-)metamodel, Ecore is self-defined, which means it is its own metamodel. Fig. 2.9 shows a simplified subset of the Ecore metamodel.

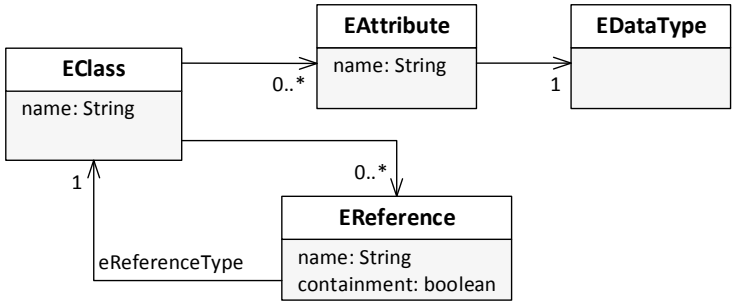



Figure 2.9: Simplified subset of the Ecore metamodel (adapted from Steinberg et al. 2008)

[14]http://eclipse.org/modeling/emf/
[15]http://omg.org/spec/XMI/
[16]http://jaxenter.com/eclipse-modeling-framework-interview-with-ed-merks-10027.html

As expressed by the boolean attribute `containment` in Ecore's `EReference` class, references in EMF-based metamodels can be *non-containment references* (also called *cross-references*), or *containment references*, that is, aggregations. Aggregation defines an ownership relation between one model element, the container, and the referenced element. Containment implies a number of constraints for models that must be ensured at runtime, in other words, for the objects which represent a model's elements at runtime. The MOF specification states that *"an object may have at most one container"* and that *"cyclic containment is invalid"*. EMF implements the first constraint by defining a single-valued back-reference to a model element's container in every EMF generated class. If, at runtime, an object is added to a containment reference, its container back-reference is automatically set accordingly, and, if this object was contained by another object the old containment relation is deleted. Thus, containment induces tree structures, called *containment hierarchies*, within a model.

Because there can be multiple elements in a model which are not contained by any other element, and therefore constitute a root of a containment hierarchy, a MOF-based model, in general, is a graph that has a spanning forest of containment trees. In EMF, however, the containment hierarchy of a model is used for the automatically provided XML-based persistence functionality of EMF-based tools. Because of the way EMF handles persistence, a third constraint is practically (but not formally) induced for EMF-based models: *"There is a distinguished object, the root object, which contains (transitively) all other model objects"* (Biermann et al., 2008). This constraint is in line with the XML well-formedness constraint which says that every XML document can only have one root element. For this reason, persisting an EMF-based model to XML does not work properly when there are multiple root elements in a model. With XMI persistence, multiple root elements are generally supported because a fake XML root element is created if needed. Still, all root elements of a model need to be added separately to a persistence resource, which is cumbersome.

Therefore, as an important practical assumption for this dissertation, we assume that every EMF-based model has exactly one element which is the root of the model's containment hierarchy. *Thus, we assume that an EMF-based model is a graph (with a reference to a metamodel) which has a spanning containment tree.*

### 2.3.2 Generated Java Types and Element Creation in EMF

The Java code which is generated by EMF from a metamodel realizes a clear separation between interface types and implementation types. For every metamodel class – say `Person` – EMF generates a Java interface called `Person` and an implementation class called `PersonImpl` which implements the interface. The generated interface always extends – directly or indirectly – the basic interface `EObject` which is EMF's equivalent to Java's base type `Java.lang.Object`. This interface provides a few basic methods for model navigation, modification, and persistence. It also enforces the aforementioned single containment by its single `eContainer()` containment back-reference accessor method which only returns `null` when called with the containment hierarchy's root object. Furthermore, `EObject` extends a notification interface which is used by EMF to realize the

Model-View-Controller pattern. The generated types follow the *JavaBeans* standard to provide public accessor methods – getters and setters – for each field of a class.

EMF models and programs processing EMF models are supposed to work in different environments transparently (e.g., with EMF models stored in file system resources, and with EMF models stored in a database). This requires that object creation can be changed easily. Therefore, EMF model elements should never be created directly via the new operator. Instead, the responsibility for object creation is given to according factories. The returned instances are only exposed via the generated interfaces. The actual implementation class type is not exposed. This application of the factory pattern allows different implementations, and gives EMF more control over configuring objects after creation (e.g., configuration as normal objects or proxy-objects, setting meta-information, etc.). Thus, an EMF object is usually created by a call to an appropriate factory method without any parameters and then filled with data using the public accessor methods.

## 2.4 The Scala Programming Language

In this section, we briefly introduce Scala concepts which are particularly helpful for developing type-safe internal DSLs. An internal DSL reuses its host language's concrete syntax. Thus, every transformation which we describe with the model transformation languages which we develop in this dissertation is a valid Scala program. If not stated otherwise code listing in this dissertation show Scala code. Therefore, basic knowledge of Scala's syntax is helpful (but not required) for reading this dissertation. Fortunately, Scala's syntax is intentionally close to the syntax of Java.

### 2.4.1 Java Interoperability

A Scala program is compiled to standard Java bytecode which can be executed by any Java Virtual Machine (JVM). Thus, Scala is a *JVM language*. The virtual machine cannot distinguish between bytecode generated from a Scala program, a Java program, or by any other JVM language. Therefore, one can mix bytecode generated by Java and Scala in the same JAR file. Many Scala IDEs such as the *Eclipse* Scala IDE plug-in allow Java and Scala source code files to be mixed in one project and to be compiled (and type-checked) together. Importantly, libraries written in Java – such as EMF – can be accessed from Scala seamlessly while keeping static type-safety. Scala libraries can be accessed nearly seamlessly from Java: implementation details of some of Scala's high-level abstractions and its syntactic sugar is exposed when accessing from Java because these concepts have to be represented in Java's generally lower abstraction level and with Java's less advanced type system. In general, the relation between Java and Scala has similarities to the relation between C and C++ (although Scala was never planned to be a strict superset of Java). C++ was designed to add features for object-oriented programming to C; Scala was designed to add features for functional programming to Java.

### 2.4.2 Flexible Syntax and Type Inference

Scala's syntax resembles Java with three major exceptions. First, Scala permits omitting semicolons, dots in method invocations, and parentheses when a method is called with only one argument. `"Hello".charAt(1);` can be written as `"Hello" charAt 1`. Therefore, when using suitable method names, statements can resemble natural language sentences. Secondly, type annotations are optional in most cases and follow the identifier (as in UML). Instead of their type, method definitions begin with `def`, immutable variable definitions with `val`, and mutable variable definitions with `var`. Their type can be inferred in most cases, while still providing static type-safety. Thirdly, type parameters of generic types are enclosed in square brackets (in contrast to Java's angled brackets). Array (and list) items are accessed with normal round parentheses. Finally, Scala supports many special characters in method names which enables 'operator overloading' as demonstrated with method name ° in line 4 of the following listing.

```
1 class Container[T] { // type parameters are enclosed in square brackets
2   val numbers = List(1,2,3) // type of 'numbers' is inferred as List[Int]
3   var content: List[T] = null // type cannot be inferred from null
4   def °(i: Int): T = { return content(i) }// type annotation :T & 'return' optional here
5 }
```

### 2.4.3 Function Objects and the Fluent Interface Pattern

In Scala, functions are objects. Functions can be created anonymously, the syntax is `(arg: T) => {block}`. In the following listing the functions `arithmeticMean` or `geometricMean` can be passed for an easy to read invocation of `calculate` as demonstrated in line 8.

```
1 object Calculator { // a simple internal DSL defined by chainable methods
2   def calculate (fnc: (List[Int]) => Int) = { ...; this }
3   def geometricMean(lst: List[Int]): Int = { ... }
4   def arithmeticMean(lst: List[Int]): Int = { ... }
5   def of(lst: List[Int]) = { ... }
6 }
7 // using the calculator DSL resembles natural language:
8 Calculator calculate arithmeticMean of List(1,2,3)
```

The example DSL also shows the application of the *fluent interface* pattern, which is often used for internal DSLs. Method `calculate` returns the calculator object (`this`) so that method `of` can be invoked immediately afterwards without having to refer to `Calculator` again. Such a fluent interface can be implemented in many languages. However, in Scala it is particularly effective to create the feel of an independent DSL because of the possibility to omit dots and parentheses in method invocation. The keyword `object`, used in the first line of the above listing, creates a singleton object which is the standard way to define class methods in Scala – it is the equivalent to defining methods using the `static` keyword in Java.

### 2.4.4 Implicit Conversions

Usually, one can only change or extend own code. For example, adding a new method to the existing `java.lang.String` class is not possible. Languages like Ruby and Smalltalk circumvent this: they allow modifying a class for the whole application. Scala provides *implicit conversions* to change the perceived behavior of a class in a given scope. Implicit conversions are methods annotated with the additional keyword `implicit`. The implicit method `implicit def fromAToB(from: A): B = new B(from)`, for example, converts an object of type $A$ to an object of type $B$. With this implicit conversion declared or imported, objects of type $A$ can be used as objects of type $B$ within the current scope. The Scala compiler will simply insert an invocation of this conversion method as needed. For example, if type $B$ provides a method named `foo` and type $A$ does not, the statement `a.foo()` is implicitly augmented at compile-time to `fromAtoB(a).foo()`. Importantly, this process is type-checked. If there is no conversion found in scope that has a suitable input and output type, a compile-time error will occur. For transparency, the compiler-augmented code that includes the inserted conversion calls can be viewed using a special compiler option. Also, many Scala IDEs support highlighting of implicit conversions. Still, extensive usage of implicit conversions can be confusing at times.

### 2.4.5 Case Classes and Pattern Matching

Using the `case` keyword in front of a class definition enables a very succinct class declaration syntax, as shown in lines 2–3 of the following listing: Immediately after the class name follows the constructor parameter list which, at the same time, determines public fields of the case class. A case class can inherit one class and multiple *traits*. A trait is similar to an interface in Java, which means multiple inheritance from traits is allowed. However, case classes cannot be inherited themselves. Every case class automatically provides a number of convenience methods which enable instantiation without `new`, instance comparison using `==`, `<=` etc., and *pattern matching*.

```
1  sealed trait BoolOrChar // a root type of a case class type hierarchy
2  case class B(b: Boolean) extends BoolOrChar
3  case class C(c: Char) extends BoolOrChar
4
5  def makeStringFrom(x: BoolOrChar) = x match {
6    case B(true) => "true"
7    case B(false) => "false"
8    case C(c) => c.toString
9    case null => "null"
10   case _ => "?" // compiler warns that the default case can never be reached because the
11 }               // match is already exhaustive, i.e., all possible cases are covered
12 val str = makeStringFrom(B(true)) // testing the method with a new instance of class B
```

As shown in lines 5–11, the pattern matching syntax '`case` *pattern* `=>` *return value*' allows a concise implementation of alternative handling that would otherwise require multiple nested if-else clauses. One can match for the type of a case class instance, its member values, or for null values. Moreover, when matching for case class instances that inherit from a *sealed* trait, which means that all subclasses are known at compile-time,

the compiler can both statically check whether all possible patterns are matched and whether there are specified cases which can never be reached.

### 2.4.6 Type Parameters, Type Bounds, and Type Argument Inference

Scala's type system provides advanced means for defining type-generic classes and methods. The following listing shows the declaration of a generic class with two type parameters, `Elem` and `Container`. The second type parameter `Container` has one type parameter itself which is here specified to be the same type as `Elem`. Furthermore, type parameter `Container` is restricted to be a subtype of type `Traversable`, which is specified using an *upper bound* with the syntax '*subtype* `<:` *supertype*'. There are other type bounds, e.g., a *lower bound* `>:` which restricts a type to be a supertype of another, or a *view bound* `<%` which restricts a type to be representable as another in the sense that there is a suitable implicit conversion in scope.

The generic class provides one generic method `firstOf` with one type parameter `T` which is restricted to be a subtype of the class' type parameter `Elem`. The method's value parameter *c* is specified to be of type `Container[T]`, that is, of the type of the class' type parameter `Container` type-parameterized with the method's type parameter `T`. Because the method uses method `head` defined by `Traversable`, the return type of `firstOf` is inferred as `T`.

```
1 class HelperMethodsFor[Elem, Container[Elem] <: Traversable[Elem]]() {
2   def firstOf[T <: Elem](c: Container[T]) = c.head
3 }
4 val valueListHelper = new HelperMethodsFor[AnyVal, List]() // explicit type arguments
5 val intList = 7 :: 8 :: 9 :: Nil // idiomatic Scala for creating a list of integers
6 val fst = valueListHelper.firstOf(intList) // type argument is inferred as Int
```

Now in line 4 an instance of the generic helper method class is created and its type arguments are explicitly specified as `AnyVal` and `List`. Thus, the created instance provides helper methods for lists whose elements inherit from `AnyVal`, i.e., lists of value type elements like `Int`, `Boolean` etc. In line 6, the `firstOf` method of this instance is called with a list of integers. According to the defined type bounds, the compiler would neither allow an invocation with a list of strings nor with a set of integers. Importantly, the method's type argument does not need to be specified explicitly because it can be inferred as `Int` from the passed value argument, which is of type `List[Int]`.

This type is also inferred. The list creation statement in line 5 uses the singleton object `Nil` which represents the end of a list and provides a generic right-binding prepend method named '`::`'. This method infers the type of its value argument '9' to be `Int` and therefore returns a new list of integers with one element. This returned list object of type `List[Int]` provides a similar `::`-prepend method. However, this method is not generic because the list elements' type `Int` is already set by now. Thus, the subsequent prepend operations which add the values 8 and 7 to the list are checked to only accept integers.

# 3 Model Synchronization in a Domain-Specific Workbench

In this chapter, we analyse what kind of transformation languages are required for model synchronization in multi-view domain-specific workbench built from modelware tools. Therefore, we first present the *NanoWorkbench*, a domain-specific workbench which we developed in cooperation with a group of physicists. The *NanoWorkbench* serves us as a practical example for model synchronization in domain-specific workbenches in general. As it turns out, there are quite different model synchronization tasks. Therefore, in section 3.2, we present a taxonomy for categorizing model synchronization scenarios. Notably, in Sec. 3.2.2 we precisely define the term 'model synchronization'. In section 3.3, based on the *NanoWorkbench* and the taxonomy, we describe what transformation language features are required for implementing model synchronization in a modelware-based domain-specific workbench. Based on these requirements, two model transformation languages are developed in the subsequent chapters.

This chapter is partly based on material which has been published in Wider et al. (2011), Wider (2011), and Diskin, Wider, Gholizadeh, and Czarnecki (2014).

## 3.1 The NanoWorkbench – A Workbench for Experimental Physics

The motivation for the work presented in this dissertation emerged from the development of a domain-specific workbench for a group of physicists. In the following sections, we first present the domain of simulation-driven development of optical nanostructures. Afterwards, we present the developed languages and tools. Thereafter, we discuss model transformation challenges, in particular those imposed by view synchronization.

### 3.1.1 The Domain: Simulation-Driven Nanostructure Development

In general, *optical nanostructures* are structures that are smaller than the wavelength of visible light, which is why they can affect the behaviour of light. More specifically, the long-term goal of research into optical nanostructures is to produce *photonic components* whose features are similar to those of electronic components. Because photonic components use photons instead of electrons for information transmission, photonic components could have advantages over today's electronic components, e.g., less heat development. Thus, a computer made of photonic components might be able to operate at higher frequencies than today's computers. Furthermore, photonic components are also important for the development of quantum computers.

Of particular interest are periodic optical nanostructures, so called *photonic crystals.* Photonic crystals are designed to affect the motion of photons in a similar way that semiconductor crystals affect the motion of electrons. An example of a photonic crystal is shown in Fig. 3.1. As can be seen, the shown structure is a thin membrane with a lattice of holes in it. In the middle of the structure some holes were omitted.

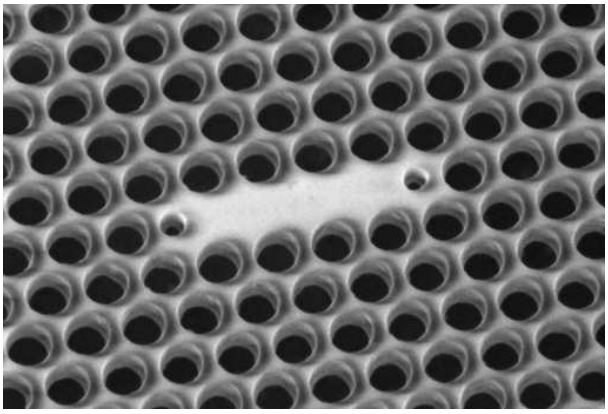


Figure 3.1: A photonic crystal (from Barth et al., 2007)

In simulation-driven development of photonic crystals, typically, the propagation of an electromagnetic pulse is simulated. In the case of the example structure above, the simulation shows that resonances occur where holes were omitted, which is the desired behaviour of the structure (Fig. 3.2). The area where resonances occur is called *cavity* because light is captured in it.

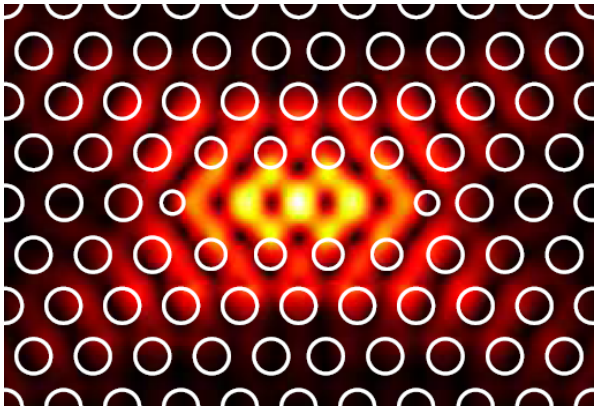


Figure 3.2: Result of simulation shows resonances (from Barth et al., 2007)

There are different simulation methods and different implementations of simulation methods that differ in accuracy as well as in resource consumption. Therefore, it is desirable to be able to flexibly choose the simulation method and implementation which fits best for the research question at hand. In the context of our research cooperation, two different implementations of the *finite differences time domain method* (FDTD) were

mainly used: the commercial tool *Lumerical FDTD Solutions*[1] and the open-source tool *Meep*[2], which both come with their own set of tools to describe an experiment and to parameterize simulation.

The general workflow in simulation-driven physics – as shown in Fig. 3.3 – is as follows: First, the geometry of the nanostructure is described. Second, the simulated experiment is described, e.g., the source of the electromagnetic pulse is specified, and the simulation is parameterized. Third, the simulation is performed and produces results which need to be analyzed, either manually or automatically. Depending on the outcome of this analysis either the experiment setup, the simulation parameters, or the geometry of the structure are modified. This loop is usually repeated several times. Finally, when the simulation shows satisfying results, these results are verified by producing a real photonic crystal and by performing a comparable real-world experiment with it. In the particular research group, the specification for this real-world experiment was usually made by hand as the used simulation tools provide only limited export features. Because one expensive real-world experiment is preceded by a series of (comparably inexpensive) simulations, we call this approach to nanostructure development *simulation-driven*.

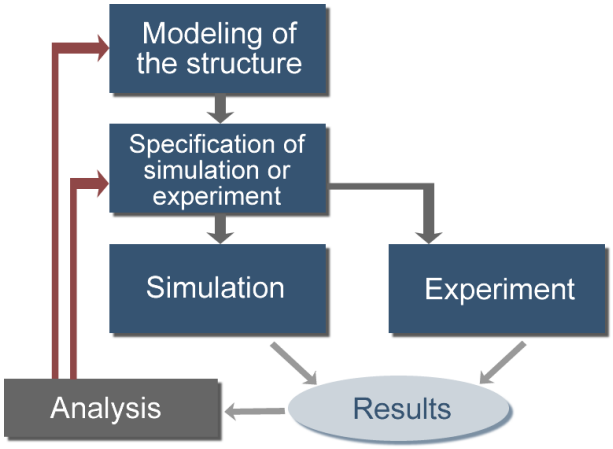



Figure 3.3: Nanostructure development workflow

### 3.1.2 Applying MDE to Nanostructure Development

Before applying MDE to this workflow, the physicists were facing several problems. Although conceptually the description of an experiment in most simulation tools is very similar, the concrete tool-specific descriptions differ. For example, in *Lumerical*, experiments are described using either a complex, dialog-intensive GUI, or a tool-specific imperative scripting language. In *Meep*, in contrast, the experiment is described using the functional programming language Lisp.

Because of insufficient interoperability between simulation tools, it was hard to switch between different simulation methods or implementations in order to benefit from their

[1] http://www.lumerical.com/fdtd.php
[2] http://ab-initio.mit.edu/wiki/index.php/Meep

individual strengths. Furthermore, tool-specific experiment descriptions hinder knowledge transfer between research groups.

Therefore, the main goal of applying MDE to this workflow was to allow for a tool-independent and easy-to-read (for domain experts) experiment description, while being able to use existing tools for simulation. Thus, we developed a domain-specific language – the *NanoDSL* – tailored specifically to the needs of designing photonic crystals. We described the DSL using the *Xtext* language workbench, so that a rich-featured textual editor for the DSL could be generated from the DSL's description.

We developed the DSL iteratively, driven by demands from the domain experts. I.e., first we asked them how they would describe a simple experiment textually to a colleague. Based on their example experiment description, we developed an early prototype of the language and, importantly, a working tool set and gave it to the domain experts to get feedback. We call this approach to tool and language development *example-driven* (Bąk et al., 2013). Based on feedback, we modified the DSL, generated new language tooling and gave it to the domain experts again. After several feedback loops, when the simple DSL met the demands, we asked for more complex experiment descriptions and extended the DSL accordingly. Being able to generate working language tooling in every iteration helped tremendously with this agile, example-driven development of the language.

Thus, the first application of MDE is to generate language tooling from models which describe the DSL. The second application of MDE is to generate simulation tool-specific experiment descriptions from the simulation tool-independent models which are created using the DSL. Therefore, a code generator was implemented as a set of model-to-model and model-to-code transformations. Fig. 3.4 illustrates our application of MDE to nanostructure development, and how code is generated from models at different meta-layers.

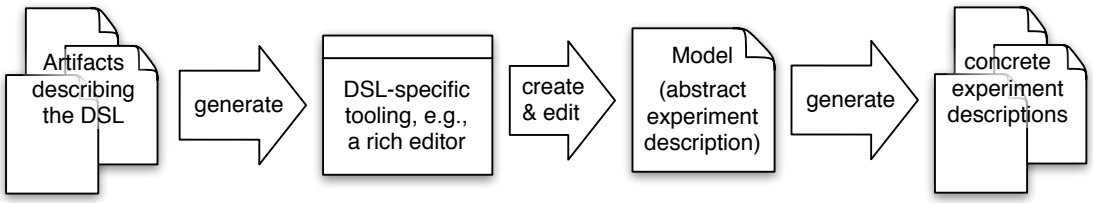



Figure 3.4: Applying MDE to the development of optical nanostructures

In the following two subsections we first present the *NanoDSL* and its tooling, and afterwards the implementation of code generation for targeting different simulation tools.

### 3.1.3 The NanoDSL: A Textual Language for Describing Experiments

The *NanoDSL* is a textual DSL, tailored to the needs of the domain experts. The general decision for a textual concrete syntax was also based on discussions with the domain experts. A textual syntax is better suited for expressing mathematical expressions. Also, there are many tools for managing text files, e.g., for version control and comparison. We use *Xtext* because it is specifically designed to create textual DSLs and their tooling. In

the following subsections we explain the *NanoDSL*'s domain-specific concepts, how it is described using *Xtext*, and present the language tooling generated from this description.

### Concepts and Structure of the Language

An experiment description in simulation-driven nanostructure development is divided into four main parts: (1) a description of the structure itself, i.e., its material and its geometry, (2) the simulation parameters such as resolution and time, (3) placement and properties of electromagnetic sources, and (4) the specification of *monitors* which define what information is to be collected during simulation, e.g., a two-dimensional plane in the three-dimensional simulation space.

The domain experts' typical approach to the description of the geometry of a photonic crystal is as follows: First, the parameters of a periodic lattice of holes are defined, and afterwards, modifications to that lattice are described. Thus, the starting point is always a flat cuboid which represents the membrane, and a lattice of holes within this cuboid. The parameters for the lattice are defined as follows: First, the alignment of the holes is specified: either rectangular (90 degree) or hexagonal (60 degree). Next, the distance between the holes and their radii have to be defined. Finally, the number of holes is set by specifying a two-dimensional array. Fig. 3.5 shows the rectangular and hexagonal arrangement and the two main lattice parameters: hole radius (r) and hole distance (d).

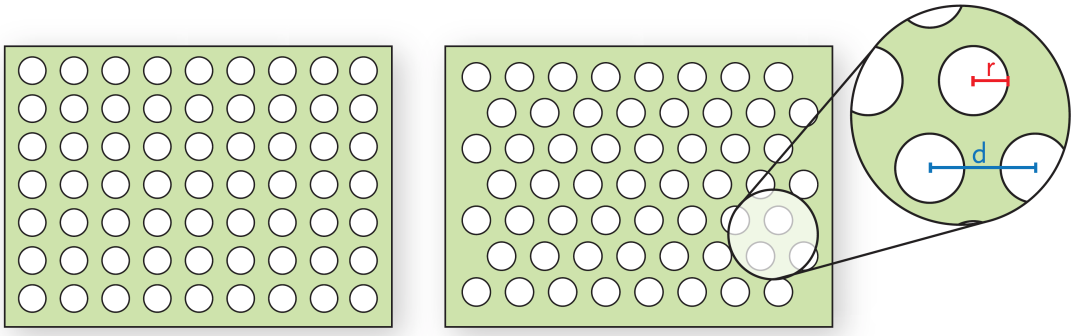



Figure 3.5: Orthogonal (left) or hexagonal (right) lattice setup (from Schmidt, 2011)

For the next step of the structure description, means for modifying the lattice of holes are provided. For convenience, the *NanoDSL* provides means to select and modify a single hole as well as a selection of lines or ranges of holes. The earlier defined array of holes serves as a two-dimensional coordinate system whose origin is located in the middle of the lattice. Each hole – being identified by its coordinates – can be deleted, moved or overwritten by another geometrical object. Furthermore, it is possible to add other geometrical objects, and to define different materials to alter the standard setup. Fig. 3.6 exemplifies several ways of modifying the lattice by deleting a selection of holes from it, and shows the concrete syntax for the corresponding operation in the *NanoDSL*.

In the *simulation part*, parameters of the simulation are specified, for instance, the simulation space can cover the whole nanostructure or only a selected region. Further-

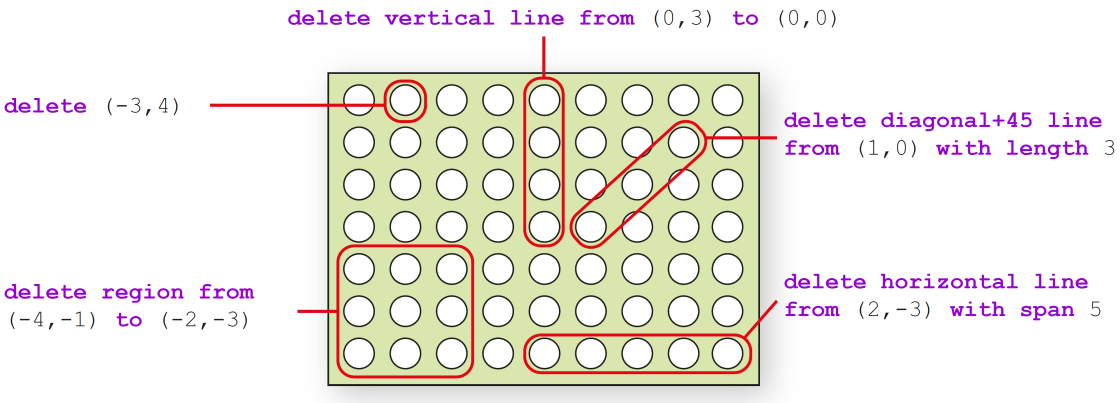



Figure 3.6: Delete edit operations for modifying the lattice of holes (from Schmidt, 2011)

more, the simulation time and the resolution of the simulation are defined. The settings made in the simulation part affect the resource consumption of the simulation the most.

The next part of the experiment description defines *sources* of electromagnetic pulses that are propagated within the photonic crystal. A source can be either a narrowband or a broadband pulse. Additionally, the position and direction of the pulse can be defined.

The last part defines *monitors*, which determine what data is to be collected during simulation. We identified three types of monitors: a *box monitor* which collects data in a given three-dimensional space, a *frequency-domain field monitor* which analyzes data over a given frequency spectrum, and a *point monitor* which focuses on a single point.

Listing 3.1 shows the structure part of an experiment description in the NanoDSL's concrete textual syntax. The described structure is the photonic crystal which we showed before in Fig. 3.1. The keywords of the *NanoDSL*'s textual syntax were defined according to the domain-specific vocabulary of the domain experts, for example, the cuboid representing the thin membrane is called a *slab*. There are no units used in the experiment description because they are either implicit – for instance, the standard length unit in that domain is nanometer – or they are explicitly defined globally for the complete model.

### Describing the Language, Starting With the Concrete Syntax

As we explained in Sec. 2.2.2, for a metamodel-based language the metamodel is the pivotal artifact of the language description, and concrete syntax and semantics are defined with respect to the metamodel. For example, it is described how elements of the concrete syntax are mapped to elements of the abstract syntax.

Because *Xtext* is specialized on textual languages, their concrete textual syntax is usually described first, namely by a grammar, and then the metamodel is automatically generated from that grammar. We call this the *concrete-syntax-first* approach. Listing. 3.2 shows a small part of the grammar which describes the *NanoDSL*'s concrete syntax, and which indirectly also describes the *NanoDSL*'s abstract syntax. The rules of the grammar are described in an EBNF-like grammar-description meta-language provided by *Xtext*.

In Xtext's grammar description language, the name of a non-terminal production rule

Listing 3.1: The structure part of an experiment description with the *NanoDSL*

```
SETUP { ... } // omitted here; contains project name etc.
STRUCTURE {
  material GaP = 3.3 // the refraction index of gallium phosphide

  Slab { // defining the cuboid representing the membrane
    material = GaP // referencing the material defined above
    thickness = 70.0
  }

  Lattice { // defining the lattice of holes
    lattice_type = hexagonal // 60 degree alignment
    lattice_size = (19,17)
    lattice_distance = 209
    hole_radius = 0.4 * 200
    // modifications to the holes surrounding the cavity
    overwrite region from (-4,1) to (4,-1) {
      Cone { radiusGround = 65; radiusTop = 65 }
    }
    overwrite region from (-4,0) to (4,0) {
      Cone { radiusGround = 55; radiusTop = 55 }
    }
    move (-4,0) { x_offset = -0.3 }
    move (4,0) { x_offset = 0.3 }
    // the actual cavity
    delete horizontal line from (-2,0) to (2,0)
  }
}
SIMULATION { ... }
SOURCES { ... }
MONITORS { ... }
```

starts with an uppercase letter (e.g. `Model` in line 1), and the rule's body follows after a colon. As the most important difference to an EBNF grammar, non-terminals inside a rule's body can be preceded by an identifier and a '=' to indirectly define fields in the classes of the generated metamodel. In line 2, the `Model` rule refers to the `SetupSec` rule, and at the same time defines the field `setupSec` in the generated metamodel class `Model`. A multi-valued field – and at the same time non-terminal repetition – is defined by (*field*+=*Rule*)+ or (*field*+=*Rule*)*, respectively, depending on whether the field is allowed to be empty or not. Importantly, an abstract rule which only consists of alternative non-terminals, for example the `Objects` rule in lines 23–24, results in an abstract class in the metamodel that is inherited by the classes which represent the alternatives. Fig. 3.7 shows parts of the metamodel generated from the grammar. One can see that models which conform to this metamodel are graphs, e.g., class `Slab` contains a non-containment reference to class `Material`. However, the containment relations form a spanning tree within this graph, with an instance of class `Model` being the root of the containment tree. This is enforced for EMF-based models in general, and for metamodels generated by *Xtext* in particular.

From the concrete syntax description and the metamodel generated from it, *Xtext* can automatically generate a rich-featured textual editor for the *NanoDSL*. Features

Listing 3.2: A part of the grammar describing the *NanoDSL*'s concrete syntax

```
Model:
  setupSec=SetupSec
  structureSec=StructureSec
  simulationSec=SimulationSec
  sourceSec=SourceSec
  monitorSec=MonitorSec;

SetupSec:
  "SETUP" "{"
    (values+=Instance)*
    "project_name" "=" projectName=STRING
  "}";

StructureSec:
  "STRUCTURE" "{"
    (material+=Material)*
    (objects+=Objects)+
  "}";

Material:
  "material" name=ID "=" index=NUMBER;

Objects:
  Slab | Lattice| Element;

Slab:
  "Slab" "{"
    (("position" "=" coordinate=Coordinate3D)
    &("material" "=" material=[Material|ID])
    &("thickness" "=" thickness=Calculation) )
  "}";
```

provided by the editor include syntax highlighting, code completion, error highlighting and an outline view which supports code navigation (Fig 3.8).

### 3.1.4 Model Transformation: Code Generation and Multiple Views

With the *NanoDSL* and its generated editor, domain experts can describe an experiment at a high level of abstraction using domain-specific syntax and vocabulary. However, in order to integrate existing simulation tools and to allow multi-view editing in the *NanoWorkbench*, the experiment model has to be transformed. In the following subsections, we discuss these model transformation tasks.

#### Targeting Existing Simulation Tools

Our application of MDE to nanostructure development is not only concerned with specification and documentation of experiments but also about model execution, which in this case means performing a simulation of the described experiment. The experiment model abstracts from the concrete way an experiment is performed, for instance, which simulation tool is used to run a simulation or how an according real-world experiment

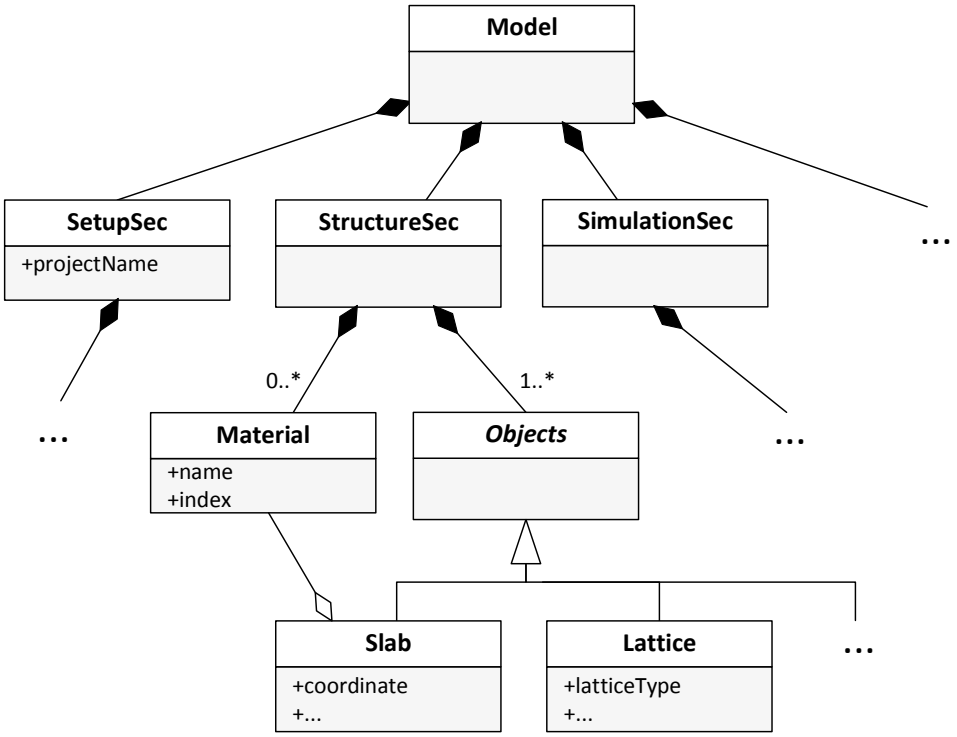



Figure 3.7: The *NanoDSL*'s generated metamodel consisting of Java classes and interfaces

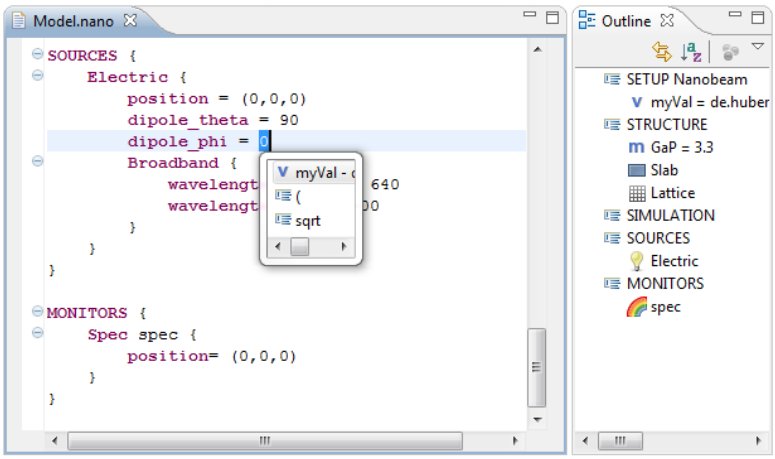



Figure 3.8: The generated textual editor for the *NanoDSL*

is set up. In order to use existing simulation tools, valid input for these tools needs to be generated from the experiment model. Both *Lumerical FDTD Solutions* and *Meep* accept text files as input which have to contain *Lumerical script* or Lisp code with *Meep* library calls, respectively. Code generators targeting these tools can be implemented by model-to-code transformation. Conceptually, every code generator is a different, indirectly defined, execution semantics of the *NanoDSL*.

When we implemented these code generators we experienced that they were conceptually very similar. We decided to merge common parts of these code generators into one model-to-model transformation. This transformation produces an intermediate model which is conceptually closer to the input required by simulation tools. However, the intermediate model still abstracts over the different input formats of the different simulation tools. From this intermediate model, much simpler model-to-code transformations are required to target different simulation tools. This not only reduces redundancy between code generators but also allows more static verification of transformations to be applied. In contrast to a model-to-text transformation which produces untyped text, the output type of a model-to-model transformation is explicitly defined by its target metamodel, so that the well-formedness of the transformation output can – at least partly – be statically checked. Thus, implementing code generation as one shared model-to-model transformation and several small model-to-code transformations reduces both the effort to integrate further simulation tools as well as the likeliness to have mistakes in the code generator implementation.

The arrangement of model transformations in the *NanoWorkbench* is illustrated in Fig. 3.12 on p. 56. The implementation of this shared model-to-model transformation is part of our case study and therefore presented in detail in Sec. 6.1 of Chap. 6.

#### Multiple Views

Although the domain experts liked the conciseness of the textual *NanoDSL*, they were used to having a visualization of the geometry of the designed nanostructure from their simulation tools. A visualization of the geometry helps to detect mistakes and provides a quick overview of the nanostructure. To provide a geometry visualization view as part of the *NanoWorkbench*, we could reuse the model-to-model transformation presented in the previous section. The intermediate model produced by this transformation is already much closer to a declarative geometry description needed for a visualization than the model created directly with the *NanoDSL*. Therefore, providing a *passive* – that is, not editable – geometry visualization view is simple. Fig. 3.9 shows a 2D visualization view which accompanies the textual *NanoDSL* editor. The view displays the result of creating a lattice of holes according to the lattice specification and of applying all specified lattice modification operations. The view is refreshed every time the textually described *NanoDSL* model is saved, by triggering the model-to-model transformation.

However, as the geometry view is a useful abstraction of the experiment description – a perspective focusing on the geometry – certain tasks can be easier achieved by allowing edits directly from that view, such as moving or deleting single holes. This means that edits made in the geometry view need to be propagated back to the *NanoDSL* model which

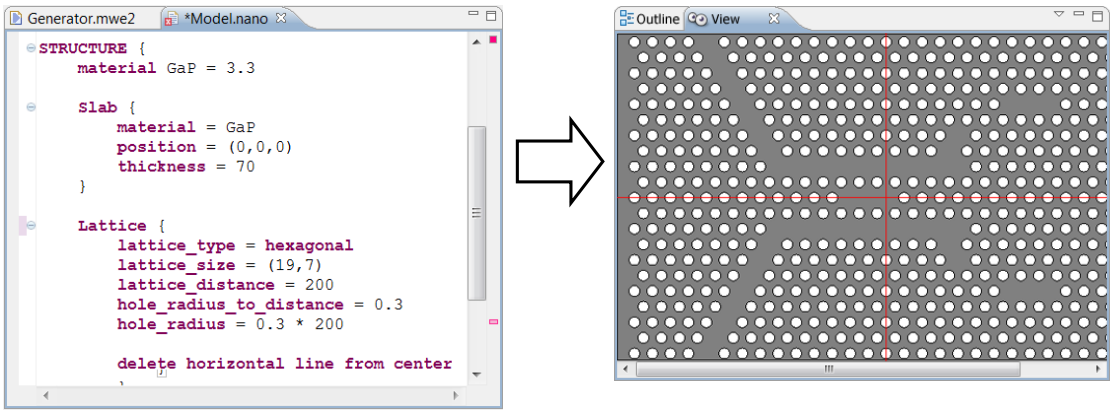



Figure 3.9: The nanostructure view visualizes the geometry of the nanostructure

captures the complete experiment description. Editing a system description from several task-specific perspectives is called *multi-view modeling* or *multimodeling* (Hessellund et al., 2007). As already discussed in Chap. 1, providing multi-view editing capabilities in a domain-specific workbench can be achieved by bidirectional model transformation. The implementation of a bidirectional model transformation for an editable geometry view is part of our case study and presented in detail in Sec. 6.2 of Chap. 6. However, realizing view synchronization by bidirectional model transformations is not the only approach to multi-view modeling. In the next section, we discuss different approaches to multi-view modeling and explain that we chose a model transformation based approach because it allows us to reuse existing tools for generating language tools without modification.

### 3.1.5 Approaches to Multi-View Modeling

There are different approaches to multi-view modeling in a domain-specific workbench. One approach is implementing different views as *multiple concrete syntax* – e.g., textual and graphical ones (van Rest et al., 2013). A view maps an element of the abstract syntax to its concrete representation and displays it. Multiple views can display different representations of the same abstract syntax element, which therefore acts as a common underlying model of these views. Interacting with an *editor* – i.e., an editable view – manipulates this common underlying model directly (Fig. 3.10). There is no need of complex inter-view synchronization because other views just have to be refreshed when a change has been made on that common model. This is similar to the well-known model-view-controller software design pattern.

However, this approach is only easy to implement if the relation between the structure of a concrete syntax representation and the structure of a model can essentially be described by a bijection; for example, if the keywords of a textual concrete syntax can be mapped one-to-one to the classes of the metamodel. If the relations between a model and its representations are bijective, also the relations between those representations must be bijective. This approach to multi-view modeling is therefore only easy to implement if all editable views display representations which are essentially bijectively related. If their

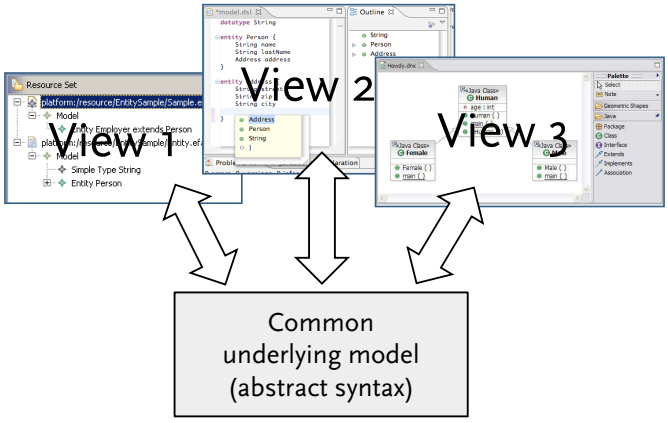



Figure 3.10: View synchronization with multiple concrete syntax

relations are not bijective, implementing the logic to update a view or to propagate edits made to the view back to the common model gets complex and possibly hard to maintain. However, language workbenches which implement projectional editing (presented in Sec. 2.2.6.) apply the multiple-concrete-syntax approach and show that it is also feasible with non-bijective relations.

Avoiding complex non-bijective concrete syntax mappings, several existing modelware technologies which allow the automatic generation of language tooling, such as *Xtext* and *GMF*, are designed for a bijection between metamodel elements and elements of the concrete syntax description. In the case of *Xtext* this one-to-one relation is obvious because the metamodel is usually generated from the concrete syntax description. Therefore, it is difficult to combine the multiple-concrete-syntax approach with those existing modelware tools for editor generation without modifying them.

An alternative approach to multi-view modeling which allows the reuse of existing technologies such as *Xtext* is to provide each view with its own underlying model for a simple concrete syntax mapping, and to synchronize these models with model-to-model transformations (Kalnina and Kalnins, 2008) This *mappings-and-transformations* approach to multi-view modeling is illustrated in Fig. 3.11. Obviously, the complexity of non-bijective relations is simply moved from the concrete syntax mapping to these model synchronizations. However, with this approach technologies such as *Xtext* can be used without any modification because they do not even need to know that they are used in a multi-view context. They just get a notification for refresh when their model has been synchronized with the other underlying models. As explained in Sec. 1.1, synchronizing the underlying view-models can be achieved either with pairs of unidirectional model transformations or – in order to increase maintainability – using bidirectional model transformation.

### 3.1.6 The NanoWorkbench as a Network of Models & Transformations

As we have seen in the two previous subsections, both breaking down code generation into several steps and providing multiple generated editors, lead to additional models with their own metamodels as part of the workbench, beside the main experiment model

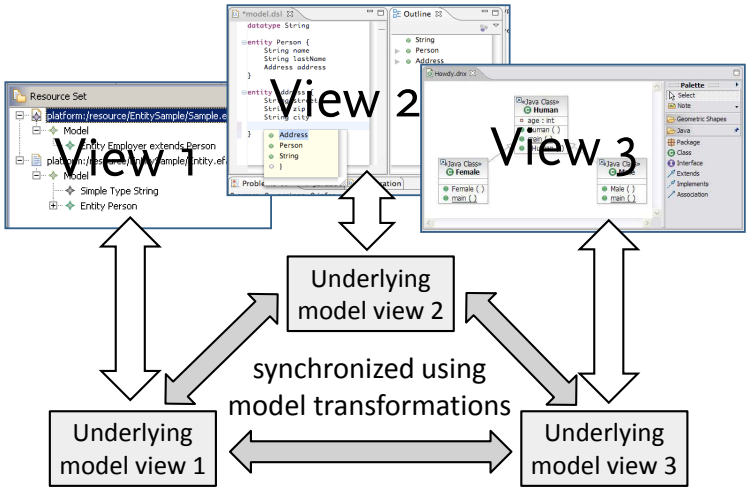



Figure 3.11: View synchronization approach with *mappings* (white) between a view and its underlying model and *transformations* (gray) between those models.

and its metamodel. These models are related to the experiment model either directly or indirectly by model transformation. In the *NanoWorkbench*, all editors are generated from a language description: a metamodel describing the structure of the underlying view model, a concrete syntax description such as a grammar, and a transformation-based semantics description relating the view model to other models in the workbench. As such, the *NanoWorkbench* can conceptually be interpreted as a *network of heterogeneous models* which are connected by model transformations.

As we have illustrated in the previous subsections, these transformations are of different kinds: There are model-to-model and model-to-text transformations, there are unidirectional and bidirectional transformations. There might also be partly bidirectional transformations, e.g., when a view allows only certain operations whose effects can be propagated to other models, as it is the case with the outline view of the *Eclipse JDT*. Furthermore, there are bijective relations implemented by model transformations and there are non-bijective relations. Non-bijective relations can be surjective, e.g., when synchronizing a view such as the geometry view which only presents selected information with the central *NanoDSL* editor which presents the complete experiment description. Fig. 3.12 shows the models which are subject to transformation in the *NanoWorkbench* and their different relations.

This situation of a network of models related in various ways is substantially different from the code-generation centric MDA scenario where mainly unidirectional transformations are applied to generate executable code from high-level platform-independent models. *In a domain-specific workbench, it is not software that is mainly being modeled, but domain-specific knowledge which is described in various ways and needs to be kept consistent.* In order to manage this complex situation of a heterogeneous network of models, it is required to have a good understanding and categorization of the different ways in which models can be related. The following section provides such a categorization.

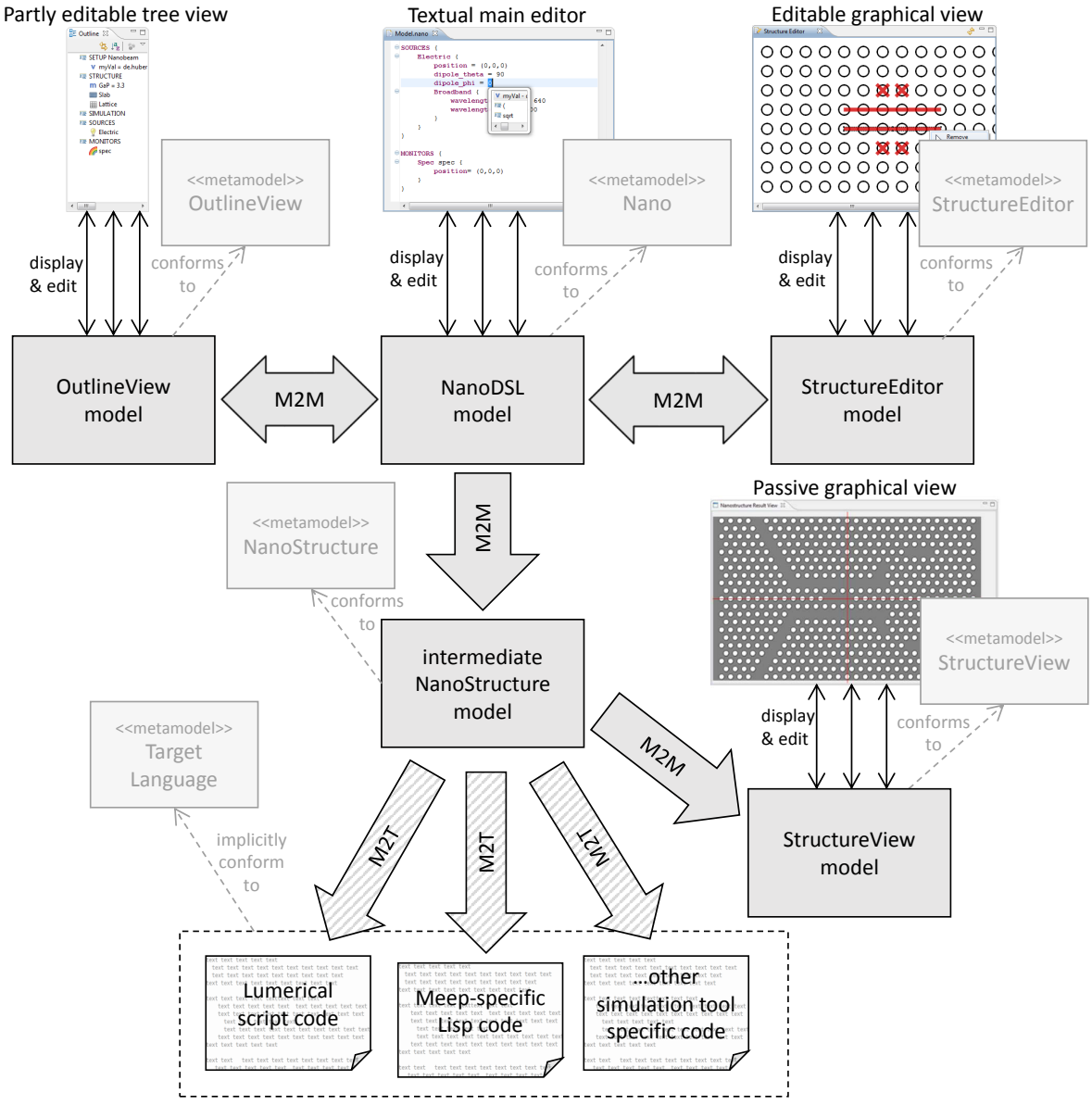



Figure 3.12: The *NanoWorkbench* as a network of models and transformations

## 3.2 A Taxonomic Space for Increasingly Symmetric Model Synchronization

In this section, we construct a taxonomic space of model synchronization types. The section is structured as follows: First, we introduce the general idea of symmetrization as a trend from pipelines of unidirectional transformations to networks of interrelated models. Next, we define what we mean by model synchronization. We then present two orthogonal concepts along which model synchronization types can be categorized: organizational and informational symmetry. We then present incrementality as a third orthogonal feature, so that we get a three dimensional space of synchronization types. Afterwards, we illustrate the symmetrization trend by presenting a series of model synchronization scenarios and locate each scenario in the taxonomic space. Finally, we discuss challenges posed by symmetrization and how our taxonomic space can help.

### 3.2.1 From Transformation Pipelines to Networks of Models

Early MDE was based on a simple generic scenario promoted by the MDA approach: platform-independent models describing a software system at a high-level of abstraction are transformed stepwise to platform-dependent models, from which executable source code is automatically generated. The generated code was meant to be a secondary artifact which could be discarded anytime such as generated assembler or byte code, whereas models were the primary artifacts to be maintained.

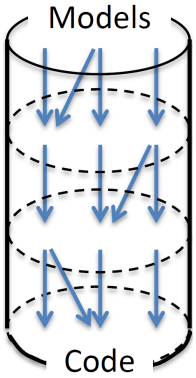
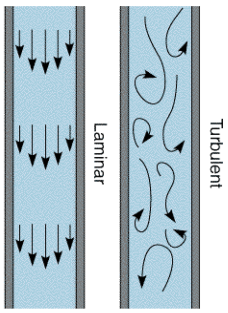


Figure 3.13: MDA pipe

Software development in the MDA perspective appears as a collection of model-transformation chains or streams "flowing through the MDE-pipe" where all transformation goes unidirectionally from higher to lower levels of abstraction, as shown in Fig. 3.13. However, this pipeline architecture fails to capture two important aspects of practical MDE. First, it turns out that some changes are easier to realize in lower-level models (including code) rather than in high-level models. This requirement leads to round-trip engineering in which transformation-streams in the MDE-pipe flow back and forth. Second, models on the same or different abstraction levels are typically overlapping rather than bering disjoint, which in our pipe analogy means that transformation-streams interweave rather than flow smoothly. Round-tripping and overlapping thus change the flow from "laminar" to "turbulent", as illustrated by the inset figure on the right. Instead of separated and weakly interacting transformation-streams, we have a network of intensively interacting models with bidirectional horizontal (the same abstraction level) and vertical (round-tripping) arrows as shown in Fig. 3.14.




"Turbulency" of modern model transformation leads to several theoretical and practical challenges. Semantics of turbulent model transformation is not well understood,

whereas clear semantics is crucial for synchronization tools because otherwise users have no trust in automatic synchronization. Furthermore, tool users and tool developers need a common language to communicate required and provided features because not every synchronization problem requires the same set of features, and implementation of unnecessary features can be costly and increases chances of unwanted interaction. Having a taxonomy of synchronization behaviors, with a clear semantics for each taxonomic unit, would help to manage these problems.

We analyze the basic unit of a model network – a pair of interrelated models to be kept in sync – and build a taxonomy of relationships between two models from the viewpoint of their synchronization, assuming that concurrent updates are not allowed. It is a strong simplifying assumption; however, this setting already covers many cases of practical interest and the presented concepts provide a basis for investigating the more complex concurrent change scenarios. We identify three orthogonal dimensions in the space of such relationships, and 21 synchronization types – points in the space. The space equips this multitude of types with a clear structure: every type is characterized by a triple of its coordinates, which together determine its synchronization behavior. We will also show that synchronization types can be ordered by having more or less symmetry in their behavior. Then the evolution of MDE from its early pipeline setting to its current state can be seen as a trend through the space from asymmetric to symmetric synchronization types. Therefore, we call this trend *symmetrization* (and illustrate it in Fig. 3.17 on p. 66).

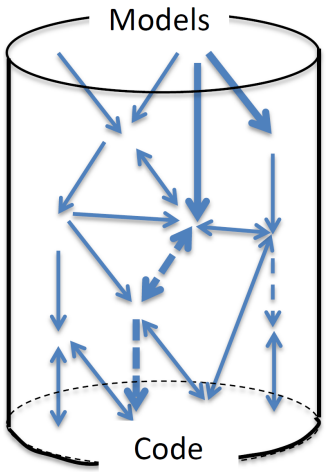


Figure 3.14: Model network

In Diskin et al. (2014), we presented a sketch of an algebraic framework in which our synchronization types are formally defined. Algebraic laws related to a synchronization type induce requirements to synchronization procedures realizing the type. Hence, classifying a concrete synchronization case by its type helps to identify and communicate the right specification and the right tool for the synchronization problem at hand.

### 3.2.2 What is Model Synchronization?

When different models overlap in the sense that they contain the same information in different ways – as it is the case with the view models in *NanoWorkbench* – their consistency must be ensured when any of them changes.

*Definition* 3.1 (heterogeneous model synchronization, model synchronization). *Heterogeneous model synchronization* is the process of establishing consistency within a set $S$ of models which conform to different metamodels. Consistency is defined by a consistency condition $c$ which is a function that takes $S$ and maps it to either true or false.

Synchronizing a set of models can be realized by synchronizing its elements pair-wise until all elements are consistent with each other. Therefore, from now on, we mainly look

at the synchronization of a pair of heterogeneous models (*binary heterogeneous model synchronization*) as the smallest unit of synchronization. Here, a consistency condition can be described more specifically. Consistency of two models $a$ and $b$ which conform to metamodels $\mathcal{M}$ and $\mathcal{N}$, respectively, means that $\langle a, b\rangle$ is an element of a consistency relation $C_{\mathcal{MN}} \subseteq M \times N$, where $M, N$ are the sets of all possible models that conform to $\mathcal{M}$ and $\mathcal{N}$, respectively. Thus, a consistency condition for a heterogeneous set of models can be defined by a set of binary consistency relations.

**Theorem 1.** *Let $S = \{m, n, o...\}$ be a set of models which conform to at least two different metamodels in a set $MM = \{\mathcal{M}, \mathcal{N}, \mathcal{O}...\}$. Let $C = \{C_{\mathcal{MN}}, C_{\mathcal{MO}}, C_{\mathcal{NO}}, ...\}$ be a set of binary consistency relations. A consistency condition $c$ over $S$ can be defined by $c = \forall\ x, z \in S : P(x, z)$ with $P$ being defined either directly as $P(x, z) \equiv \langle x, z\rangle \in C_{\mathcal{XZ}}$, where $x$ conforms to a metamodel $\mathcal{X}$ and $z$ conforms to a metamodel $\mathcal{Z}$, or transitively, e.g., with $P(x, z) \equiv \langle x, y\rangle \in C_{\mathcal{XY}} \ \wedge\ \langle y, z\rangle \in C_{\mathcal{YZ}}$. In general, $c$ can be defined using the transitive closure of the union of all consistency relations: $c = \forall\ x, z \in S : P(x, z) \equiv \langle x, z\rangle \in (\bigcup C)^+$. The number of required consistency relations $|C|$ depends on the number of different metamodels $|MM|$: it is at most $\frac{|MM| \cdot (|MM|-1)}{2}$ (all directly related) and at least $|MM| - 1$ (most transitive relations).*

So now that we know how to describe consistency, how can we enforce consistency – i.e., realize synchronization – by means of model transformation? A unidirectional model transformation can be seen as a function on models. It describes a binary relation of source and target models. If a (unidirectional or bidirectional) model transformation implements a consistency relation, the transformation can be used to establish consistency.

*Definition* 3.2 (binary heterogeneous model synchronization, binary synchronization). Let $\langle x, y\rangle$ be a pair of inconsistent models which conform to metamodels $\mathcal{X}, \mathcal{Y}$, respectively, where consistency is defined by a consistency condition $c$ which is defined by a binary consistency relation $C_{\mathcal{XY}}$. Then *binary heterogeneous model synchronization* is an operation which alters one or both models, so that they are consistent with respect to $c$.

The simplest case of synchronizing two inconsistent models is to choose one of them – the one whose changes are to be preserved – and use a unidirectional model transformation which implements the consistency relation to create a (by definition) consistent model. The originally inconsistent model is then replaced with the newly created model, discarding all information in the original model. Thus, any model transformation can be considered as a model synchronization.

When synchronizing a pair of models, we can distinguish different general situations which we call *synchronization types*. Two concepts that are helpful for categorizing binary synchronization scenarios are explained in the following subsection.

### 3.2.3 Organizational and Informational Perspectives on Synchronization

In this section, we consider two basic features of binary model synchronization scenarios: *organizational symmetry* (org-symmetry) and *informational symmetry* (info-symmetry),

and then discuss their orthogonality and the 2D-plane formed by their combination. Org-symmetry is fundamental for model synchronization but, to our knowledge, has not been discussed in the literature in technical or formal terms. Org-symmetry captures the idea of one model being more authoritative than the other. Info-symmetry characterizes "equality" of informational contents of models. This feature, and its phrasing in terms of symmetry, is well known in the literature on algebraic models of bidirectional transformations (Diskin et al., 2011b,a; Hofmann et al., 2011).

**Organizational Symmetry**

Suppose that two models to be synchronized, $a$ and $b$, are given together with a consistency relation between them. Assume that $a_1$ and $b_1$ are two inconsistent states of the models, and one of the models, or both, are to be changed to restore consistency. There may be different policies for such changes. As we discussed before, a simple one is when one of the models (say, $a$), is considered entirely dominating the other model $b$. That way, consistency restoration is realized by changing $b_1$ to $b_2$ which is then consistent with $a_1$. This situation is common when a low-level model $b$ (e.g., bytecode) is generated from a high-level model $a$ (Java program). Generating Java code (this time model $b$) from a UML model $a$ is similar, if round-tripping is not assumed. The low-level model $b$ is not supposed to be modified manually. When the high-level model $a$ is changed, the original low-level model is discarded and a new model is regenerated from scratch. In all such cases, we say that model $a$ *organizationally dominates* $b$, and write $a{>}_{\mathsf{org}}b$. Equivalently, we say that $b$ is *dominated by* $a$ and write $b{<}_{\mathsf{org}}a$. We will also refer to these cases as *organizational asymmetry*.

We have an entirely different synchronization type for code generation with round-tripping. Suppose that a UML model $a_0$, and a Java program $b_0$ which was generated from $a_0$, were initially consistent, but later model $a$ was changed to state $a_1$ inconsistent with $b_0$. Then program $b_0$ must be changed to $b_1$ which is consistent with $a_1$. We say that update $a_0 \to a_1$ on the $a$-side is propagated to the $b$-side. Conversely, if model $b_0$ (code) was changed to $b_1$ inconsistent with $a_0$, then model $a$ must be changed to restore consistency, and we say that update $b_0 \to b_1$ was propagated to $a$. Thus, in contrast to organizational dominance, propagation can go in either direction based on the history: the freshly updated model dominates irrespectively to whether this freshly updated model is either $a$ or $b$. We say that neither model organizationally dominates the other, write $a \asymp_{\mathsf{org}} b$, and call this situation *organizational symmetry*. The basic question characterizing the organizational dimension is the following: *In what direction are updates propagated?* Are they propagated only from $a$ to $b$, only from $b$ to $a$, or in either direction?

There are also important synchronization cases in-between the strict asymmetry and strict symmetry considered above. A model can be *partially dominated* in the sense that *some* (but not all) updates on this model are allowed to propagate to the other side depending on the type of the update. Consider, for example, the outline view of *Eclipse JDT*. The outline view is regenerated every time the Java code changes. Thus, there seems to be an organizational dominance of the Java code over the outline view. However, the JDT outline view allows the user to make some selected operations in the outline view,

e.g., renaming elements, or moving elements within the hierarchy. These updates are then propagated to the code. So, whereas all updates from the code (model $b$) are propagated to the abstract outline view $a$, only selected updates on $a$ are allowed, and thus can be propagated, from the outline $a$ to the code $b$. We call this situation *organizational semi-symmetry* and write $a \leq_{\mathsf{org}} b$ (note the difference from $a <_{\mathsf{org}} b$ denoting asymmetry). A similar semi-symmetric variant of code generation from UML models could be also constructed. In contrast to the strict asymmetry version discussed above (no changes from code $b$ are propagated to model $a$, $a >_{\mathsf{org}} b$), some code updates – for example, changes in method heads – are allowed to be propagated to the model. We will therefore sometimes refer to organizational semi-symmetry as *partial round-tripping*.

Organizational semi-symmetry also includes a setting, in which both models are partially dominated, i.e., both update propagation directions are sensitive for the update type. Consider, for example, a system model consisting of a UML class diagram and a UML sequence diagram with the following synchronization policy. If a class name is changed in the class diagram, this change has to be reflected in the sequence diagram, but class name changes are not allowed in the sequence diagram. Dually, if a method signature is changed in the sequence diagram, this change has to be reflected in the class diagram, but the latter is not allowed to change method signatures. Thus, to completely characterize the organizational dimension, one has to ask: *Which updates (if any) are propagated in what direction?*

### Informational Symmetry

The notion of informational symmetry is based on inter-model consistency. The latter can be modeled as a binary relation $C \subset M \times N$, where $M$ and $N$ denote the sets of all models which conform to metamodels $\mathcal{M}$ and $\mathcal{N}$, respectively. In general, the consistency relation is of type many-to-many. For example, if $M$ is a set of UML models, and $N$ a space of Java programs, a given UML model $a \in M$ can be correctly implemented by many Java programs $b \in N$; differences between these $b$s are usually termed as "implementation details". On the other hand, $a$ normally contains some information not relevant for code generation, such as layout of boxes and arrows, timestamps, etc. Hence, the same Java program can be a correct implementation of, generally speaking, different UML models. Thus, each of the models has some *private* information not needed for the other model, and they both share some *public* information important for the other model, but represent it differently. We then write $a \asymp_{\mathsf{inf}} b$ and term the case as *info-symmetry*.

We have an essentially different synchronization situation between code and its outline view in a typical IDE, e.g., the JDT outline mentioned above. The outline only shows parts of the information that is presented in the code, or, more generally, an abstract view of the code so that only one outline model $a$ is consistent with a given piece of code $b$. Of course, the same outline model $a$ may be consistent with many implementations $b$, so that the consistency relation is of the one-to-many type. We then write $a \leq_{\mathsf{inf}} b$ and term the case as *info-asymmetry* (below we will explain why we use $\leq_{\mathsf{inf}}$ rather than $<_{\mathsf{inf}}$).

So far, we used the term 'view' informally to describe common GUI elements of domain-specific workbenches that display only selected information of a bigger model. With the

concepts of organizational and informational asymmetry, we can define the concept of a view. In order to distinguish the GUI element from the general informational concept, we will – when it is not clear – refer to the latter as the 'view model' to make clear that we only refer to the underlying information which is displayed by a GUI view.

*Definition* 3.3 (view model). A *view* is a *role* of a model in a model synchronization context. A *view* $v$ can always be uniquely extracted from another model $s$, which in this scenario is referred to as the *source model* (or just the source). View $v$ is informationally dominated, $v \leq_{\mathsf{inf}} s$, so that several source models $s_1$, $s_2$, etc. can match the same $v$, whereas $v$ cannot have private information. If $v$ is allowed to be modified, the view role implies that updates made to $v$ are supposed to be propagated back to the source $s$ immediately. Thus, $v$ acts as an interface for modifying $s$. In this case, we call $v$ an *active view*. If $v$ is not allowed to be modified, i.e., if there is $v \leq_{\mathsf{inf}} s$ combined with $v <_{\mathsf{org}} s$ so that $v$'s only purpose is to present some of the information of $s$, we call $v$ a *passive view*.

Note that info-asymmetry appears in the case of code generation, if we consider UML models up to their code-relevant context. That is, we consider two UML models equivalent, if they only differ in their concrete syntax and layout, and ignore all attributes not needed for code generation (like authorship and timestamps). Then consistency becomes a one-to-many relationship, and we have $a \leq_{\mathsf{inf}} b$ (UML model $a$ is an active view). This conceptual simplification of code generation is a useful model of the situation.

An important characteristic of info-asymmetry is that the computational nature of update propagation essentially depends on the direction. Propagating updates from the source $b$ to the view $a$ is a relatively simple computational procedure. In contrast, propagating updates from the view to the source is non-trivial because some missing information on the source side is to be restored (see Foster et al., 2007; Diskin et al., 2011b). For the info-symmetric case, both update propagation directions need restoration of missing information and both are non-trivial.

A special case of info-symmetry is when the consistency relation is of the one-to-one type and determines a bijection between two model spaces: now neither of the two models has private information, i.e., both models are just different representations of the same information. We write $a =_{\mathsf{inf}} b$ for info-bijectivity. An example of such bijective situation is synchronizing a wiki article described in a lightweight markup language like MediaWiki with the equivalent HTML description of the article. Although bijectivity is symmetric, it is convenient to consider it as a special case of info-asymmetry: each of the two models can be uniquely extracted from the other and update propagation is simple in both directions. To show that bijectivity is included into the info-asymmetry, we write $a \leq_{\mathsf{inf}} b$ rather than $a <_{\mathsf{inf}} b$. The latter can be used to explicitly exclude bijectivity and denote strict info-asymmetry where $b$ must have a private part while $a$ must have no private part.

### Organizational and Informational Symmetries Together

Recall two cases of info-asymmetry, $a \leq_{\mathsf{inf}} b$. The first is when $a$ is the outline view of code $b$. The second is when $a$ is a UML model whose syntactical peculiarities (i.e., those which

do not affect code generation) are ignored for synchronization, and $b$ is generated code. Despite the same info-asymmetry relationship between the models, their synchronization situations (we also say synchronization types) are different. Indeed, in the former case, the view is mostly a passive receiver of the source updates, and we have $a <_{\mathsf{org}} b$ (or $a \leq_{\mathsf{org}} b$, if some updates can be propagated from the view to the source). In the latter case, the view is active and generates the source which passively receives the view updates, $a >_{\mathsf{org}} b$. What determines the synchronization type of the case is a combination of two parameters indexing the organizational and the informational symmetry, respectively. Clearly, these two parameters are independent, and hence can be considered as two orthogonal coordinates forming the plane shown in 3.15.

The vertical axis has two points corresponding to the two possibilities of the info-symmetry: $a \leq_{\mathsf{inf}} b$ (y=0) and $a \not\leq_{\mathsf{inf}} b$ (y=1). The horizontal axis has three basic points corresponding to the three possibilities of org-symmetry considered in Sec. 3.2.3: $a <_{\mathsf{org}} b$ (x=0), $a \leq_{\mathsf{org}} b$ (x=$\frac{1}{2}$), and $a \not\leq_{\mathsf{org}} b$ (x=1).

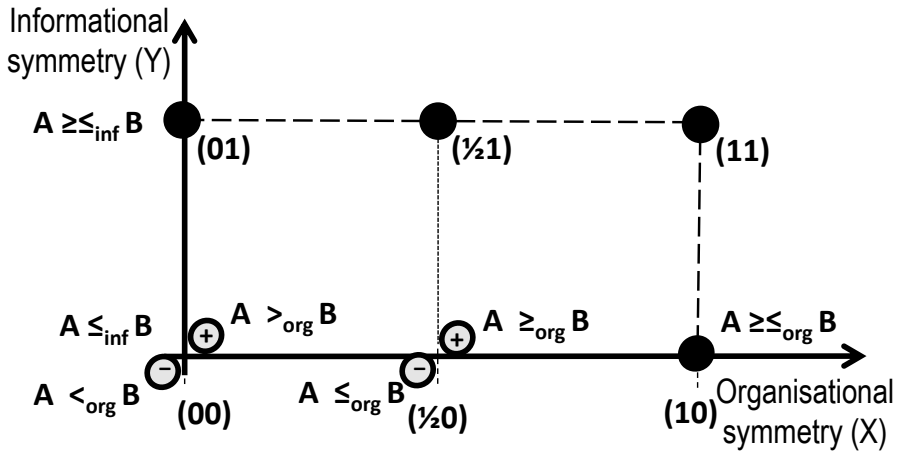



Figure 3.15: Plane of organizational and informational symmetries

However, when we consider real synchronization types, i.e., pairs (xy) on the plane, each of the types (00) and ($\frac{1}{2}$0) splits into two subtypes depending on whether two dominant models coincide or not. We will denote these subtypes by $(xy)^-$ (even less symmetry, since the same model is "suppressed" in both relations), or $(xy)^+$ (more symmetry as one model dominates in one relation, while the other model in the other relation). Thus, the plane comprises eight synchronization types. Each case considered above obtains its unique synchronization type, and each type in the plane can be interpreted by a practically interesting situation along the lines described in the section.

### 3.2.4 Incrementality: From a 2D Plane to a 3D Space

The third dimension of our taxonomy is *incrementality*, a well-recognized feature of model transformations. In the following subsections, we first discuss semantic aspects of incrementality and its connections with informational (a)symmetry. Afterwards, we show how the entire 3D-space of synchronization types is built.

### Incrementality and Delta Propagation

A *non-incremental* unidirectional model transformation $t: M \to N$ from a model set $M$ to a model set $N$ creates a new target model $b = t(a)$ from scratch every time the source model $a$ changes, no matter how big the change is. An *incremental* model transformation is supposed to be more intelligent: a small change in model $a$ is transformed into a respective small change in $b$.

In some synchronization scenarios, incrementality is optional and just improves efficiency. For example, incremental refresh of the outline view in an IDE may improve efficiency when dealing with very large code files. There are, however, situations in which incrementality is crucial and the required synchronization cannot be realized without incrementality. An example is partial code generation. Take a UML tool that generates code stubs from class diagrams, but does not support round-tripping: Code for class declarations and method heads is generated, but code in method bodies is to be added at code level. Now, when method signatures are changed in the class diagram, method heads must be regenerated while preserving method bodies, otherwise method implementations would be lost. Thus, while non-private parts of code (method heads, class names, etc.) are updated to reflect changes in the UML model, the private data of the code – the method bodies – must be preserved.

Such a situation is typical when updates are propagated to a model containing private data, if the latter is to be preserved. In more detail, an incremental transformation takes an update (delta) on one side, say, $\Delta_A: a_0 \to a_1$, and the original model $b_0$ on the other side, and produces an update (delta) on the other side, $t(\Delta_A, b_0) = \Delta_B: b_0 \to b_1$, which restores consistency between $a_1$ and $b_1$, and keeps the private part of $b_0$ unchanged in $b_1$. Deltas are ideally implemented as *traces* of what happened (or should happen) to individual model elements. If correspondences between models $a$ and $b$ are also precisely traced, an update propagation satisfying the requirements above can be assured (Diskin et al., 2011b,a; Hermann et al., 2011). We call this implementation of incrementality, where detailed traces on one side are translated to detailed traces on the other side, *delta- or trace-based update propagation.*

Another case of incrementality is when deltas are degenerated into pairs of states, i.e., $(a, a')$, $(b, b')$, etc. because not all necessary traces can be provided (e.g., updates to code are often not tracked individually). We call this *state-based update propagation* and call this implementation of incrementality *discrete incrementality.* In this case heuristics-based model-matching can be used to infer traces from those pairs. We call this process *delta discovery.* As delta discovery is needed for correct update propagation, discrete incrementality means that delta discovery is integrated with update propagation and is therefore hidden from the user. With the concept of discrete incrementality, synchronization tools can be categorized depending on whether they integrate delta discovery or not, and synchronization tasks can be categorized depending on whether detailed traces of updates can be provided or not.

Irrespectively of how incrementality is implemented, there are three ways of how a bidirectional transformation can support incrementality: no incrementality, i.e., all update propagation is non-incremental, which we denote by $a \|_{\mathsf{inc}} b$; half incrementality, e.g.,

only update propagation from $a$ to $b$ is incremental, which we denote by $a >_{\text{inc}} b$; and full incrementality, i.e., both directions of update propagation are incremental, which we denote by $a \asymp_{\text{inc}} b$. With a unidirectional transformation, however, it is of little use to distinguish between half and full incrementality because the one direction of update propagation can either be incremental or not. We therefore always consider a unidirectional transformation with incremental update propagation as fully incremental.

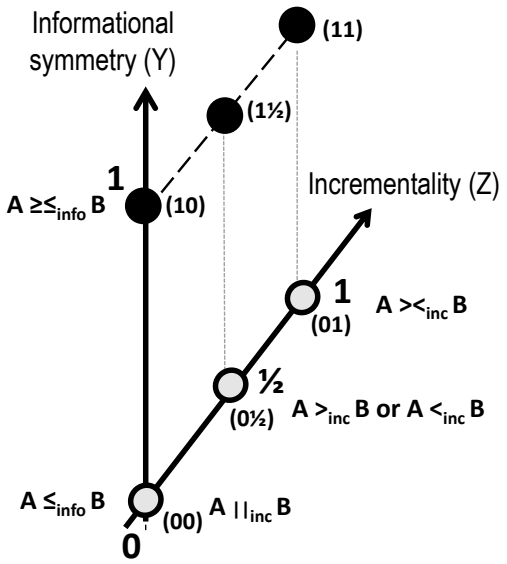



Figure 3.16: Plane of incremental model synchronization

By considering our examples of code generation from UML models, and computing an outline view by an IDE, it is easy to see that non-incremental transformation can be implemented in either direction irrespectively of info-symmetry relation between models. Moreover, correspondences between models can be established and then used for incremental (or half-incremental) synchronization also irrespectively to the info-symmetry relation. Hence, the absence or presence of incrementality can be seen as a new dimension orthogonal to info-symmetry, and together they form a taxonomic plane in Fig. 3.16.

Although incrementality can be added to any type of the info-symmetry relation between two models, the way incrementality is implemented depends on this type. The semantics of incremental synchronization depends on the info-symmetry relation because the latter determines partitioning of model's data into shared and private. This is crucial for a proper incremental synchronization. The same is true for non-incrementality as well: non-incremental code generation and external view computation are as different semantically as their incremental versions. Thus, each of the points on the plane in Fig. 3.16 determines a specific semantic framework for model synchronization. Such frameworks, we will refer to them as *computational frameworks*, can be formalized with a family of algebraic structures called *lenses* (Diskin et al., 2011b; Hermann et al., 2011; Diskin et al., 2011a). Lenses are explained in detail in Chap. 5.

### A Three-Dimensional Space of Model Synchronization Types

Clearly, org-symmetry and incrementality are orthogonal: dominance of one or another direction of update propagation, and the way the latter is implemented, can be freely combined. For example, all cases of org-symmetry discussed in Sect. 3.2.3 can be implemented incrementally or non-incrementally.

We have also seen that org-symmetry is orthogonal to info-symmetry. Hence, the org-symmetry axis X is orthogonal to the entire plane YZ of computational frameworks in Fig. 3.16, so that together they form a 3D-space as shown in Fig. 3.17. Each point in the space, i.e., a triple of coordinates, characterizes a certain synchronization behavior or *synchronization type*.

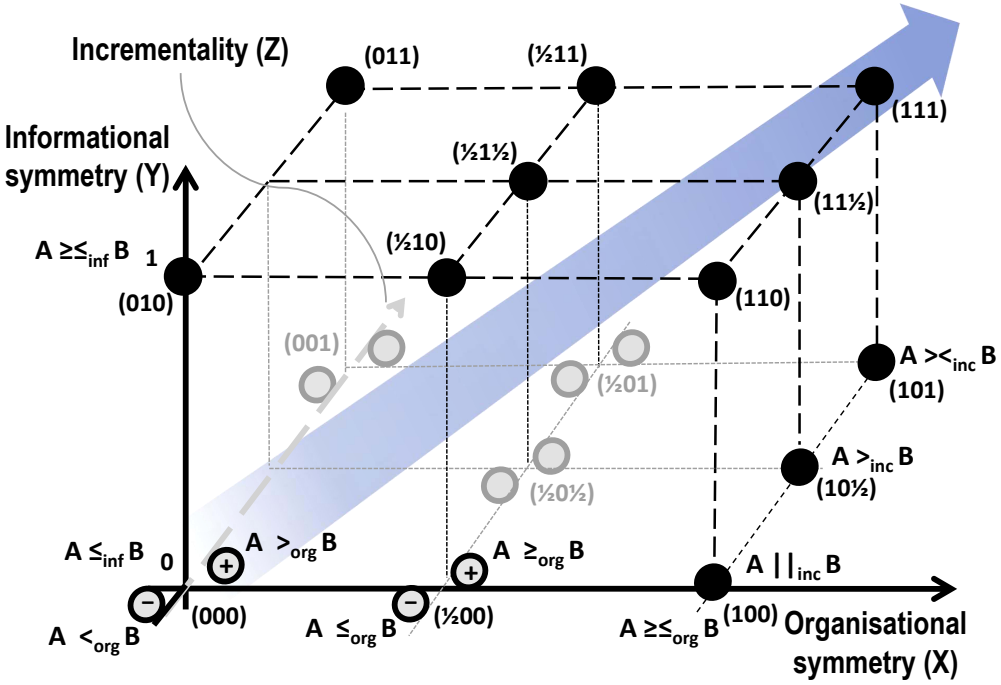



Figure 3.17: A taxonomic space of synchronization types and the symmetrization trend

Axis X is for indexing org-symmetry: asymmetry is indexed by 0, symmetry by 1, and semi-symmetry or partial round-tripping has index $\frac{1}{2}$. Axis Y is for info-symmetry: asymmetry and symmetry are indexed by 0 and 1 resp. Recall that info-bijectivity is included in info-asymmetry. For explicitly stressing info-bijectivity, an info-asymmetric synchronization type can be annotated with superscript '1:1'. Axis Z denotes whether incrementality is present or not with incr.=1, $\frac{1}{2}$, or 0, i.e., whether transformations take previous versions of models into account or execute always from scratch and whether both directions are incremental. Discrete incrementality is denoted by annotating an incremental synchronization type with $\not\Delta$ (no delta-based update propagation). Because of the split points on the bottom plane (Sec. 3.2.3, Fig. 3.15) and by excluding half-incrementality of unidirectional transformations, the space comprises $13 + 8 = 21$ distinct synchronization types. The 13 incremental types can be further differentiated depending

on whether their incrementality is implemented with or without trace-based deltas.

Every example we discussed obtains its unique type: For instance, a non-incremental passive outline view is located at type $(000)^{-}$; a symmetric multi-model specification of a system using incrementally synchronized high-level models is at type (111). Correspondingly, a synchronization tool or approach can be selected (or developed) which supports a given synchronization type. For instance, unidirectional ATL (in its standard non-incremental version) supports synchronization type (010); GRoundTram by Hidaka et al. (2011) is a tool for informationally asymmetric bidirectional transformations that supports type $(10\frac{1}{2})_{\not\Delta}$. The diagonal arrow in Fig. 3.17 is a visualization of the symmetrization trend discussed in Sec. 3.2.1; we analyze it in more detail in the next section.

### 3.2.5 Symmetrization: A Tour of Synchronization Types

With the three-dimensional taxonomic space, the *symmetrization* trend can be described as a path from simple, less symmetric synchronization types like $(000)^{+}$ to more complex, symmetric types like (111), as is visualized by the diagonal arrow in Fig. 3.17. In this section, we will illustrate symmetrization by presenting four synchronization scenarios fundamental for MDE, ordered by increasing symmetry of their behavior.

#### Model Compilation or Full Code Generation

This is the scenario envisioned, but rarely achieved, by early MDA: A high-level platform-independent model $a$ is to be edited and maintained, whereas code, model $b$, is automatically generated from $a$, taking into account platform-specific information. Model $b$ is not to be modified manually, similarly to the bytecode produced by a Java compiler. It could be optimized, similarly to bytecode optimization, but any modification will be discarded when model $a$ is changed because $b$ will be regenerated from scratch. We have $a{>}_{\mathsf{org}}b$ (x=0) because updates can be propagated only from $a$ to $b$, and $a \not\simeq_{\mathsf{inf}} b$ (y=1) because both $a$ and $b$ have private data ($a$ has layout data etc., and $b$ has platform- or implementation-specific details). Incrementality is not assumed (z=0), and the scenario thus has synchronization type (010). A conceptually useful, and often used model of the scenario is to ignore syntactical details of model $a$ and consider it as an abstract view of code $b$, which results in informational asymmetry $a \leq_{\mathsf{inf}} b$ and type $(000)^{+}$. In fact, this simplification can also be applied to the subsequently presented scenarios, so that there are multiple paths through the space which follow the symmetrization trend. Note that (irrespectively of info-symmetry) lack of incrementality combined with organisational asymmetry results in a lack of autonomy for model $b$.

#### Partial Code Generation

In practice, *partial code generation* occurred more often in early MDA, and is still common in current MDE. High-level modeling languages are often not expressive enough to allow completely automatic code generation. Then code $b$ generated from high-level model $a$ is supposed to be manually augmented with implementation details, for example, method bodies. There is no round-tripping, but in order to preserve manual modifications of

$b$, incremental model transformation is required. In practice, this incrementality has often been achieved by marking parts of generated code as protected against manual modifications. With respect to our taxonomy, we have $a >_{\mathsf{org}} b$ (updates are still to be propagated only from $a$ to $b$) and $a \succeq_{\mathsf{inf}} b$, which combined with incrementality results in type (011) or $(011)_{\not\Delta}$ depending on how update propagation is implemented. Note that although $b$ is still dominated organizationally, incrementality gives $b$ some autonomy.

#### Partial Round-Tripping

This scenario is sometimes achieved by more sophisticated MDE technologies. In comparison with partial code generation, we now allow some changes in code $b$ to be backward propagated to the high-level model $a$. However, full round-tripping is not supported yet: only some modifications in the generated code are allowed, e.g., method signatures can be modified but class names cannot. Thus, we have organizational semi-symmetry $a \geq_{\mathsf{org}} b$, informational symmetry $a \succeq_{\mathsf{inf}} b$, and incrementality, which results in type $(\frac{1}{2}1\frac{1}{2})_{\not\Delta}$ if we assume that code updates are not traced. Model $b$ gains even more autonomy but is still organizationally dominated by $a$.

#### Full Round-Tripping

All updates can be propagated in both directions. This is the (rarely achieved) goal of UML tools which promise full round-tripping with the generated code. We have $a \succeq_{\mathsf{org}} b$, $a \succeq_{\mathsf{inf}} b$, and incrementality, resulting in type (111). Now $a$ does not dominate $b$ in any way, and we have a completely symmetric situation. There is still a distinction between $a$ as a high-level model and $b$ as a low-level model, but this distinction is not captured by our taxonomy: both models have equal organizational and informational rights so that code is a first-class model as well. We therefore assume that updates to the code-model can also be traced.

### 3.2.6 Discussion: Challenges of Symmetrization

Symmetrization of model synchronization demands tools which support bidirectionality, incrementality, informational symmetry, and ultimately concurrent updates. Here, we discuss some of the challenges posed by these requirements in terms of our taxonomic space.

#### Orthogonality in Tool Architectures

Developing synchronization tools that meet all the requirements above is challenging. However, as we explained in the previous sections, several of these requirements are independent of each other, and their orthogonality can be effectively used by the tool developers. For example, it is commonly agreed that *asymmetric lenses* (Foster et al., 2007, explained in more detail in Chap. 5) implement a solution to the view update problem. With our taxonomic space, this idea of lenses can be refined. Lenses implement a *computational framework* (a point on the YZ-plane) which can be augmented with required organizational facilities (along axis X) to provide different synchronization policies

– e.g., an entirely or partially dominating view, or an entirely dominated view. Moreover, a semi-symmetric situation with both the view and the source being partially dominating (and partially dominated) is also implementable on top of the computational framework provided by asymmetric lenses. Such an extension of an existing approach is more efficient, both conceptually and implementation-wise, than developing separate tools which can only be applied to one synchronization type. In general, tool architectures that reflect feature orthogonality would allow for flexible combination of required features, and facilitate the trade-off between the synchronization capabilities and development costs.

#### Clear Semantics of Bidirectional Transformation Languages

When updates are allowed to be propagated in the two directions, two functions of update propagation, from $a$ to $b$ and from $b$ to $a$, must be consistent with another in the sense that they satisfy an invertibility property (see Sec. 5.1.1 for an example). The goal of bidirectional model transformation languages is to specify a consistency relation between two sets of models and let the two update propagation functions be inferred from this specification, so that they are always consistent by construction. As there are usually many different possibilities to restore consistency between models, the implemented behaviour of the inferred functions must be clear and predictable for the user. The current situation with *QVT-Relations* – a declarative bidirectional model transformation language standardized by the OMG – shows how unclear semantics hinders tool implementation and user acceptance. When released, *QVT-Relations* allowed the specification of non-bijective synchronizations but did not provide clear semantics for such tasks (Stevens, 2007a). It should have been either limited to bijective synchronizations – for example, type $(101)^{1:1}$ instead of implicitly suggesting support for type (111), the most challenging of all – or should have not been released without clear semantics for all supported synchronization types. With a taxonomy of synchronization types this kind of situation can be avoided. By selecting a synchronization type upfront, a synchronization tool can be designed and tailored to the requirements of that type. Furthermore, the accompanying formal framework of which we provide a sketch in Diskin et al. (2014) is meant to serve as a foundation for providing solid semantics for bidirectional languages which support informational symmetry and incrementality.

#### Concurrent Updates: Towards the Fourth Dimension

The possibility to update the two sides in parallel can be seen as an independent feature of model synchronization. Indeed, concurrency can be added to each of the org-symmetry types on axis X (including the multitude of types hidden in semi-symmetry). Of course, adding concurrency for the strictly org-asymmetric type (when one side is entirely dominated) does not make sense practically as any changes on the suppressed side will be discarded anyway, but we see it as a logically possible case, although practically not usable. Thus, each of the org-symmetry types is split into two: with concurrency allowed or not allowed, all are supplied with a computational framework. For the non-concurrent cases, the computational frameworks we considered above work without any changes, but

concurrent updates need a substantial development of their computational support. They need special procedures and policies for conflict reconciliation, and subsequent update merging (Orejas et al., 2013). Creating formal algebraic models of concurrent updates is an active research area; especially the info-symmetric case is challenging. We therefore leave adding the fourth dimension of concurrent updates for future work but kept a reminder about it in Fig. 3.17: the symmetrization arrow goes beyond the space towards even more symmetric scenarios with concurrent updates.

#### Conclusions

Symmetrization of MDE, i.e., the shift from model transformation pipelines to networks of interacting models, poses several challenges for transformation tools, such as support of bidirectionality, incrementality, informational symmetry, and ultimately concurrent updates. Developing synchronization tools which meet all these requirements is costly and implementing more features than necessary can cause unintended interactions. Having a taxonomy of synchronization behaviors, with a clear semantics for each taxonomic unit, could help to manage these problems.

We presented a taxonomic 3D-space of model synchronization types and provided it with formal semantics (sketched in Diskin et al., 2014). As far as we know the notion of organizational (a)symmetry is novel, as is its orthogonality with incrementality and informational symmetry, which have been discussed only separately or in different contexts so far. The space can be used to locate the type of the synchronization problem at hand; from this type, we can infer the requirements for model transformations tools and theories to be applied to the problem. This what we use the space for in the next section. We also think of the space as a communication medium for tool users and tool builders, in which they can specify tool capabilities and behavior. We hope that our space could also guide future research concerning bidirectional transformations, for instance by identifying synchronization types which are not covered yet – org. semi-symmetry, for example, seems particularly poorly covered. Concurrent updates are not covered yet, although we are aware of its importance for MDE applications.

## 3.3 Required Features of Model Synchronization in a Domain-Specific Workbench

The symmetrization trend in general, and the specifics of a domain-specific workbench built using modelware tools for language tool generation in particular, give rise to special requirements for model transformation tools. In this section, we gather these requirements. In general, with a network of models, managing model transformations becomes complex. Automatic tool support for correct chaining and coupling of model transformations therefore is of high importance.

In the following subsections, we first select those synchronization types of our taxonomic space which occur in a domain-specific workbench. A main issue which arises from this selection is the lack of special languages for bidirectional model transformations,

which we discuss in the subsequent subsection. We then introduce the term 'metamodel-awareness' to describe required transformation tool assistance. Afterwards, we discuss what is required to achieve technological integration with existing modelware tools.

### 3.3.1 Required Synchronization Types

In contrast to a fully symmetric scenario characterized by synchronization type (111), the scenario of a domain-specific workbench such as the *NanoWorkbench* is less symmetric because there is still a primary model, created with the *NanoDSL*. This model comprises the complete experiment description, whereas additional views only present certain aspects of the experiment description, e.g., only the geometry. However, although there are also organizational asymmetric situations, for example for code generation, the workbench scenario is still more symmetric than the entirely asymmetric MDA scenario. In a domain-specific workbench, there are editable views so that modifications need to be propagated back to the main experiment description.

In the next two subsections, we first explain why – for model synchronization in the *NanoWorkbench* – it is not required to support concurrent updates and bidirectional incrementality, and also why delta-based incrementality is not required. Then, we identify the synchronization types which are required in the *NanoWorkbench*.

#### Concurrent Updates and Full Incrementality

A domain-specific workbench is typically a single-user tool. One user can only use one view at a given time. Thus, it never occurs that multiple views are modified concurrently. Depending on how costly view synchronization is, modifications to one view can be propagated to other views at each keystroke/mouse-click, or just when the view focus changes. Thus, model transformation tools for implementing domain-specific workbenches do not necessarily need to support concurrent updates.

With a view such as the geometry view which only presents parts of the information of the full experiment description, incrementality in the direction from the view to the full description is required in order to preserve private information of the full description. Incrementality in the other direction, however, is not absolutely necessary because a view has no private part. Supporting incrementality in this direction, too – in other words providing full incrementality – could increase synchronization performance. However with a single-user tool, performance of view synchronization is not crucial and thus half-incrementality is usually sufficient in this scenario.

#### Trace-Based Incrementality vs. Synchronization-Agnostic Tool Reuse

In general, delta-based update propagation using detailed traces allows synchronization to be more efficient and – if correspondence between model $a$ and $b$ is also precisely traced – also allows change re-integration to be more reliable (Diskin et al., 2011b). However, not all modeling tools provide traces of updates. In particular, modelware tools such as *Xtext*, which we used for generating the editors which make up the *NanoWorkbench*, do not provide traces. The reason for this might be that these tools are not designed with a

non-bijective multi-view scenario in mind. *Xtext* always creates a completely new model when modifications are made in the textual editor.

Because of that, relying on traces for view synchronization inhibits the reuse of those tools without modification. In contrast, using discrete incrementality for implementing view synchronization, those editor-generating tools can be reused *synchronization-agnostically* which means that those tools do not need to know that they are used in a multi-view context. The synchronization layer just needs to recognize when a model was updated, then starts synchronization using the newly updated model together with the original target model, and after synchronization sends refresh notifications to other views.

However, model matching techniques could be used for delta discovery and could thus bridge discrete incrementality with trace-based incrementality. For instance, the synchronization layer could always store the last version of a model, so as the model gets updated, the layer would contain the old as well as the updated model. Thus, deltas can be obtained by heuristics, e.g., by using tools like *EMF Compare*[3]. This would enable the use of a synchronization method which depends on traces such as delta lenses. Of course, using heuristics instead of obtaining deltas directly from the editor, is not as reliable.

### Identifying Required Synchronization Types in the Taxonomic Space

Synchronization scenarios which are characterized by types near the right, top, back corner of the taxonomic space – near type (111) – tend to be conceptually more challenging. Therefore, we start our selection of required synchronization types in the less challenging left, bottom, front corner of the space.

Type $(000)^{-}$ (i.e., unidirectional transformations which create an informationally dominated view) needs to be supported for tasks such as providing a passive visualization view of a textually described model, or for a simple outline view displaying the hierarchical structure of the model. Type $(000)^{+}$ needs to be supported for tasks such as code generation which targets simulation tools. Here, the generated code is never modified but augmented with tool specific information which is not present in the higher-level tool-independent model. Type (010) is needed when, in the same situation, there is a private part in the high-level model that needs to be preserved – for instance, layout information – so that each side of the synchronization has a private part. As updates are never propagated back, dealing with informational symmetry is trivial in this case. A scenario with either of these three synchronization types can be implemented using unidirectional, non-incremental model transformation tools such as the *Atlas Transformation Language* (ATL).

Supporting an editable view such as the geometry editor is more challenging because edits are to be reintegrated into the full experiment description. For this, at least type $(\frac{1}{2}0\frac{1}{2})^{-}_{\mathbb{A}}$ is required. For example, when $v$ is the editable geometry view, and $s$ is the complete experiment description, we have $v \leq_{\mathsf{org}} s$ because at least some updates made to $v$ must be allowed to propagate back to $s$, whereas all changes made in $s$ are propagated to $v$. There is $v \leq_{\mathsf{inf}} s$ if we assume that unavoidable private parts of $v$ such as GUI information can be automatically generated and are therefore irrelevant for view synchronization.

[3]http://eclipse.org/emf/compare/

Finally, at least discrete half-incrementality is needed in order to allow updates to be re-integrated into $s$, hence that there is $v{>}_{\mathsf{inc}}s$ (without delta-based update propagation for easy technological integration).

Because of the orthogonality of organizational symmetry to the plane created by informational symmetry and incrementality, we can identify a *computational framework* for synchronization type $(\frac{1}{2}0\frac{1}{2})^{-}_{\not\Delta}$. The required computational framework is characterized by discrete incrementality of the forward transformation and a non-incremental backward transformation. This computational framework is realized by the concept of *asymmetric state-based lenses* as presented by Foster et al. (2005). Depending on the combination with either point $\frac{1}{2}$ or point 1 on the organizational axis, this computational framework can be used to support either type $(\frac{1}{2}0\frac{1}{2})^{-}_{\not\Delta}$ or type $(10\frac{1}{2})_{\not\Delta}$. By providing a transformation language which implements asymmetric lenses, we can at least support view synchronization with a partially editable view – e.g., one that supports a limited set of editing operations such as deleting and moving elements – and we still have the option to extend the support to a fully editable view, without having to change the computational framework.

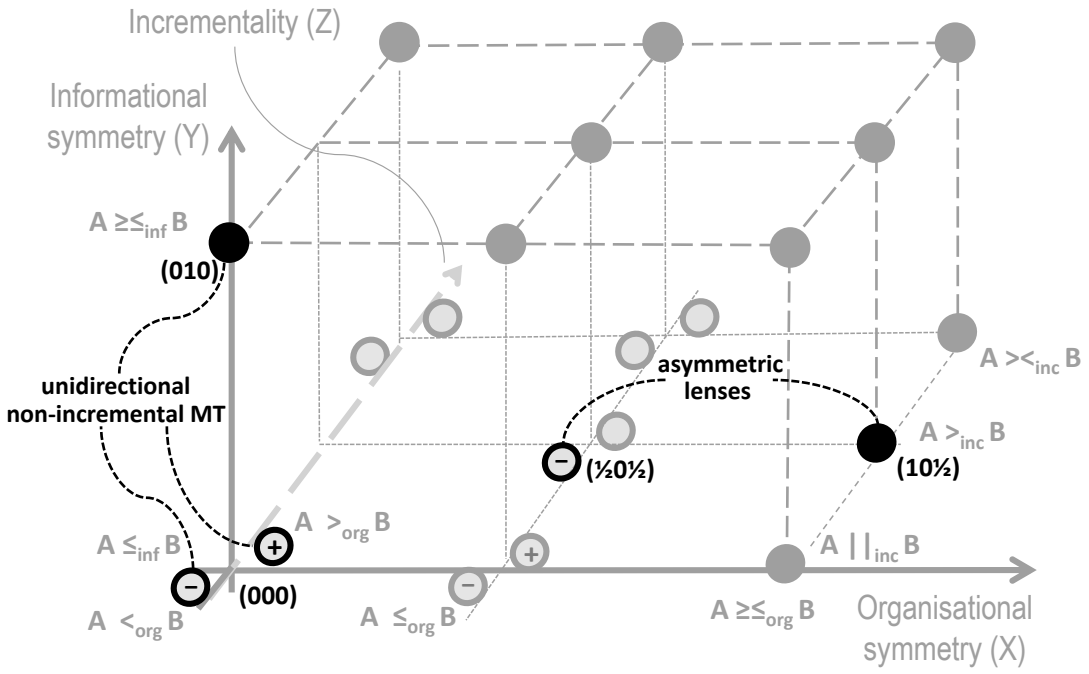



Figure 3.18: Synchronization types required by a domain-specific workbench

Fig. 3.18 shows the location of the identified synchronization types in the taxonomic space, and how they can be supported with two model transformation formalisms: unidirectional non-incremental model transformations and asymmetric state-based lenses.

Of course, it is also desirable to support more synchronization types. Within the identified five types, informational symmetry is only supported in combination with organisational asymmetry, i.e., only with unidirectional transformations (type (010)). Thus,

editors with limited or full editing support cannot have a relevant private part, for instance, they cannot keep layout information. This can be alleviated by using automatic layout algorithms, but it is still desirable to be able to preserve such additional information in views. Therefore, support for types $(\frac{1}{2}1\frac{1}{2})$ and $(11\frac{1}{2})$ is desirable but not required for transformation tools for implementing domain-specific workbenches.

We explained why neither full incrementality nor delta-based update propagation are necessarily required, and that the latter can prevent synchronization-agnostic tool reuse. However, especially when dealing with large models, both full incrementality and delta-based update propagation can increase performance of view synchronization. Thus, fully incremental types are also desirable but not necessarily required.

Other types are not needed at all. For example, type $(\frac{1}{2}0\frac{1}{2})^+$ is not needed because it is unlikely to have, for instance, generated code that is augmented with additional information but can only be edited with a restricted set of operations.

### 3.3.2 Asymmetric Bidirectional Transformation Languages

In a domain-specific workbench with editable views, there are bidirectional synchronization relations. Although in general, bidirectional synchronizations can be implemented as pairs of forward and backward transformations using any unidirectional model transformation language (or even any general-purpose programming language), this approach imposes a major maintenance issue. The consistency of the forward and the backward transformation does not only have to be shown once – which is already very difficult when several synchronizations are chained – but also the consistency needs to be ensured every time a bidirectional synchronization is modified, and both the forward and the backward transformation need to be changed separately every time.

For the implementation of a domain-specific workbench with editable views it is therefore crucial to use special bidirectional transformation languages. This allows a consistency relation to be described in such a way that the forward and back transformations can be inferred automatically, so that they are consistent by construction. Because not every transformation is unambiguously invertible, bidirectional languages provide a restricted set of special means which only allows the description of an invertible transformation. As mentioned in Sec. 3.2.6, providing clear semantics for a bidirectional language which supports informational symmetry is particularly challenging. Providing clear semantics only for informationally asymmetric synchronizations is less challenging. Bidirectional transformation languages which are restricted to the informationally asymmetric situation are called *asymmetric bidirectional transformation languages*. Because the synchronization between a domain-specific view and a bigger model, such as the *NanoDSL* model in the *NanoWorkbench*, is usually informationally asymmetric, asymmetric bidirectional languages often suffice for domain-specific workbench implementation. The usefulness of asymmetric bidirectional languages for implementing view synchronization in domain-specific workbenches has also been realized and studied by Wilson-Kanamori and Hidaka (2013).

### 3.3.3 Metamodel-Awareness

In contrast to MDA or code-generation centric MDE, generation of untyped text occurs only incidentally in a multi-view domain-specific workbench. The majority of transformations are unidirectional or bidirectional model-to-model transformations. Thus, the *typed nature* of models and model transformations (by means of metamodels) is important. In order to create and maintain a network of models and transformations, languages and tools for describing model synchronizations should be able to guarantee well-formedness of models after synchronization, and they should provide assistance based on the models' metamodels when describing model synchronizations. We call this *metamodel-awareness*.

*Definition* 3.4 (metamodel-awareness). *Metamodel-awareness* is the capability of a model transformation tool to assist the user with describing a model transformation, say $t : \mathcal{A} \rightarrow \mathcal{B}$, based on information from the source and target metamodels $\mathcal{A}$ and $\mathcal{B}$. A metamodel-aware transformation tool checks at editing time (i.e., before the transformation is executed) whether an element $e$ accessed by $t$ in a source model $a$ is specified in the source metamodel $\mathcal{A}$ and – within the limits of static verification – whether the created target model $b$ conforms to the target metamodel $\mathcal{B}$.

Based on the information from the source and target metamodels, a metamodel-aware transformation tool can also provide further assistance, for instance, by suggesting elements from the source metamodel (code completion), by inferring corresponding target metamodel elements (type inference), or by suggesting modifications to the target model, so that it conforms to the target metamodel (quick fix).

Especially in combination with declarative means for describing model transformations, metamodel-aware transformation tools can make a network of model transformations easier to understand and to maintain.

### 3.3.4 Technological Integration

For implementing model synchronization in a domain-specific workbench, suitable model transformation tools do not only need to be generally capable of describing the required synchronization tasks, but also need to integrate with the software technologies the workbench is built with. In the case of the *NanoWorkbench* all these technologies – and many DSL- and MDE-related tools in general – are based on the EMF. As explained in Sec. 2.3, an EMF-based model is, at runtime, a Java object graph which has a spanning containment tree and conforms to a metamodel which conforms to the Ecore meta-metamodel. A lack of seamless integration with EMF is often mentioned as a main reason why plain Java is still often used for implementing model transformations although Java does not provide any transformation-specific means for metamodel-aware model transformation, which is provided by special model transformation languages such as ATL.

For us, *seamless* technological integration means that EMF-based models can be transformed *directly* without the need of any import- or export-steps. Seamless integration stands in contrast to just *some* technological integration which makes interoperability *possible* but not necessarily comfortable. An example for non-seamless integration is to

indirectly allow the transformation of EMF-based models by providing an interoperability text file format as an integration layer. Seamless integration, to us, also means that it is not necessary to switch tools when working with EMF-based technologies in order to describe model transformations. An important, and here often mentioned requirement, is the capability to debug a model transformation while observing the state of the involved models – a capability that is naturally given when using Java and a Java IDE but is often not provided by special model transformation languages and their tools.

The lack of seamless EMF integration is particularly an issue with bidirectional model transformations. While there are model transformation languages for unidirectional transformations, such as ATL, which provide a decent EMF integration, there are almost no transformation languages for bidirectional model transformations which seamlessly integrate with EMF. There are powerful and actively developed tools for asymmetric bidirectional transformations like *GRoundTram* and *Boomerang*[4]. However, they originate from other technological spaces and either provide none or no seamless EMF integration.

### 3.3.5 Requirements & Assumptions

Here, we sum up the main requirements for model transformation tools and languages for implementing model synchronization in a domain-specific workbench built with modelware tools. The goal of this dissertation is to develop model transformation languages which meet these requirements. Therefore, we also infer a set of assumptions on which we can rely when developing those transformation languages. We focus on model-to-model transformations (in contrast to model-to-text transformations) because they occur more often in a multi-view domain-specific workbench. Also, there are already good tools for unidirectional model-to-text-transformations, for example, *Xtend*[5].

#### Requirements

- Means for describing heterogeneous unidirectional model-to-model transformation
- Means for describing heterogeneous bidirectional, at least info-asymmetric, model-to-model transformation with unambiguous semantics
- Metamodel-awareness concerning Ecore-based metamodels
- Seamless integration with EMF and EMF-based technologies
- No reliance on detailed traces of updates, provided by editing tools

#### Assumptions

- Models are never modified concurrently.
- Models are, at runtime, Java object graphs.
- Models have a spanning containment tree.

[4] http://www.seas.upenn.edu/~harmony/
[5] http://eclipse.org/xtend

- Models have Ecore as a common meta-metamodel (at the highest meta-layer).

## 3.4 Conclusion and Related Work

In this chapter, we presented (1) a domain-specific workbench for the domain of experimental physics, (2) a taxonomic space of model synchronization types, and (3) requirements for model transformation tools for implementing model synchronization in domain-specific workbenches.

The *NanoWorkbench* was used to gather the requirements and also serves as the case study for the evaluation. The taxonomic space is meant to serve as a means for transformation tool developers and users to identify and communicate requirements, i.e., it should help to select or develop the right tool or technique for the transformation problem at hand. Consequently, we used the taxonomic space to identify the requirements for the transformation tools which we present in following chapters. Furthermore, the taxonomic space and the identified symmetrization trend can help to identify open research questions or lack of tool support for certain model synchronization types. Thus, the taxonomic space can guide future research and tool development beyond the contributions presented in this dissertation.

### 3.4.1 Related Domain-Specific Workbench Work

There are several tools which fit in with our definition of a multi-view domain-specific workbench. A notable example which is based on the *Eclipse* platform is *Bioclipse*, a workbench for bioinformatics (Spjuth et al., 2007). However, *Bioclipse* and similar tools were developed manually with much effort. In order to justify development costs, the intended group of users has to be sufficiently large. By using MDE technologies which allow the automatic generation of language tooling, the development costs are small enough to justify the development of domain-specific workbenches for narrow application domains and a comparably small user base.

Furthermore, there are many workbenches which are specifically tailored for a subdomain of the software engineering domain. Some of these workbenches have been developed using language workbench tools, too. A notable example is *mbeddr* (Vöelter et al., 2013). It is a workbench tailored for the embedded software development domain and it is based on *JetBrain*'s *MPS* language workbench. In contrast to *Xtext*, which we used for creating the *NanoWorkbench*, *MPS* realizes the projectional editor approach. *MPS* itself is customized, extended, and configured, to meet the specific needs of the domain. Because of that, Völter et al. call their domain-specific workbench a domain-specific *instantiation* of a language workbench. In *mbeddr*, bidirectional transformations are not applied because there are no multiple editable views. Instead there is one editor which displays different context-specific notations embedded into another.

The use of bidirectional transformations in order to synchronize views in a domain-specific workbench, however, has also been studied in the bioinformatics domain by Wilson-Kanamori (Wilson-Kanamori and Hidaka, 2013). However, this workbench again has not been developed using a language workbench but was created mostly manually.

Summing up, to the best of our knowledge, the *NanoWorkbench* seems to be one of the first domain-specific workbenches for a domain outside of the software engineering domain that has been developed using a language workbench.

### 3.4.2 Related Model Synchronization Taxonomies

Existing works on model synchronization – practical and theoretical – usually focus only on one of the dimensions of model synchronization or on one specific synchronization type. For instance, the computational framework of lenses presented by Foster et al. (2007) is info-asymmetric, half-incremental, and state-based. Info-symmetric, state-based lenses were studied by Hofmann et al. (2011). Lenses with delta-based incrementality were proposed for info-asymmetry and info-symmetry (Diskin et al., 2011b,a). The org-symmetry dimension has only been discussed indirectly as the distinction between unidirectional and bidirectional transformation (Antkiewicz and Czarnecki, 2007). However, we present a more fine-grained distinction between unidirectional and bidirectional relations by introducing organizational semi-symmetry. In the community of the Triple Graph Grammars (TGGs), the org-asymmetric case (unidirectional transformation) as well as the org-symmetric case (bidirectional transformation) have been discussed (Schürr and Klar, 2008). Incrementality and concurrency of TGGs have been discussed by Giese and Wagner (2009) and Hermann et al. (2012, 2011), and by Golas et al. (2012) and Ehrig et al. (2007), respectively.

However, there is little related work which describes the combination of several dimensions of model synchronization and provides a formal foundation. The work by Antkiewicz and Czarnecki (2007) is closest to our taxonomy as it classifies different synchronization scenarios using feature modeling. That work takes into account operational aspects of synchronization for identifying 16 synchronization scenarios which can be located in our space, too. However, our work goes further by providing a formal descriptive background for each synchronization type in our taxonomic space (sketched in Diskin et al., 2014).

As far as we are aware of it, we were the first to arrange types of synchronization situations in a multi-dimensional space. Also the dimension of organizational symmetry and the associated trend of symmetrization were first considered by us.

# 4 A Rule-Based Language for Unidirectional Model Transformation

As we have seen in the previous chapter, both unidirectional and bidirectional model transformations are required in a domain-specific workbench with editable views. We also explained that metamodel-awareness and EMF integration are required features of tools for implementing model synchronization in that specific scenario. Because there is a lack of bidirectional model transformation tools which fulfill these requirements, the development of a suitable bidirectional transformation language is the foremost goal of this dissertation. Our approach for this is to *embed model transformation languages (MTLs) as internal DSLs in Scala* (refer to Secs. 2.2.5 and 2.4). Essentially, we use Scala's Java interoperability for achieving EMF integration and Scala's advanced static type checking capabilities for achieving metamodel-awareness.

However, although a bidirectional MTL implemented as an internal Scala DSL is our ultimate goal, in this chapter we first approach the development of a unidirectional MTL. This has several reasons: First, the topic of bidirectional MTLs is so intricate that it is more comprehensible to first explain our approach by implementing a unidirectional MTL. Second, our approach results in a level of EMF integration and metamodel-awareness which in certain cases exceeds that of existing unidirectional MTLs, while requiring less development effort. Third, by implementing both a unidirectional and a bidirectional MTL using our approach, we want to show its general applicability. Finally, implementing both a unidirectional and bidirectional MTL as an internal DSL in Scala allows these MTLs to be integrated with each other. This can be used for a soft migration from unidirectional to bidirectional synchronization descriptions, as we will explain later.

This chapter is organized as follows: The next section describes our general approach and explains the rationale behind it. In section 4.2, we present a basic Scala MTL inspired by ATL. In section 4.3, we show how a more declarative syntax can be achieved by using Scala's case classes and implicit conversions. In section 4.4, we show how metamodels can be represented in Scala's type system in order to use Scala's type checking for achieving metamodel-awareness. We end the chapter with related work and conclusions.

This chapter is partly based on material which has been published in George, Wider, and Scheidgen (2012).

## 4.1 Model Transformation Languages as Internal DSLs in Scala

In this section, we explain the rationale behind our approach of implementing MTLs as internal Scala DSLs. First, we explain why we chose to develop an MTL for describing model transformations instead of using a GPL. Then, we explain why we chose to develop an internal DSL instead of an external DSL, and why we chose Scala as the host language.

### 4.1.1 General-Purpose Language vs. Model Transformation Language

A model transformation can either be implemented using a general purpose programming language (GPL) or using an MTL, that is, a DSL which provides special means for describing model transformation. Examples for MTLs are ATL (Jouault and Kurtev, 2006), QVT-Operational, ETL[1] from the Epsilon language family, and Tefkat[2]. As with DSLs in general, the goal of an MTL is to be particularly expressive in its domain. Specially tailored MTLs are often more concise and expressive than a GPL when describing a certain kind of model transformation. Furthermore, the limited means of an MTL often allow automatic reasoning and static analysis to be more extensive.

Although there are powerful MTLs such as ATL, Java – a GPL – is still one of the most used languages for describing model transformations. Some of the often mentioned[3] reasons for this are:

- weak tool-support for MTLs in comparison to the powerful and rich-featured IDEs for popular GPLs such as Java
- hesitation to learn and to stay up-to-date with a new language and its tooling instead of using an already familiar GPL and its tooling
- limited means of an MTL in comparison to the versatility of a GPL
- limited or missing EMF integration in comparison to Java's natural ability to interoperate with EMF's generated Java classes

With our approach, we want to help in these respects. We want to provide MTLs which compare favourably with GPLs, especially Java.

### 4.1.2 External vs. Internal Model Transformation Language

As any DSL, an MTL can be implemented as an internal or as an external DSL. The main advantage of the internal DSL approach is that no or little effort has to be put into providing tooling for the MTL. Also, an important aspect is the maintenance of existing tools. For instance, if one provides an *Eclipse* plug-in for an MTL, this plug-in has to be updated with every update of *Eclipse*, in order to use it with the latest version of Eclipse.

[1] http://www.eclipse.org/gmt/epsilon/doc/etl/
[2] http://tefkat.sourceforge.net
[3] Unfortunately, there is no empirical study on the reasons for the limited acceptance of special MTLs.

An example for this is *mediniQVT*[4], one of the few tools for the bidirectional MTL *QVT-Relations*. The active development of this Eclipse-based tool stopped at some point. The latest supported version of Eclipse is version 3.7 which is long outdated by now.

With an internal MTL, one can rely on the tooling of the host language to be updated regularly; in our case, for instance, the Scala IDE plug-in for *Eclipse*[5]. New versions of the Scala IDE are provided regularly, and they also support older versions of Scala, so that it is not necessarily needed to update an internal Scala DSL to work with the most recent version of Scala, in order to take advantage of up-to-date Scala tooling. However, the tooling of an internal MTL, being that of the host language, is not tailored to the domain of model transformation. This is especially an issue with error messages which are often cryptic and expose how an internal MTL is implemented.

With an internal DSL a user cannot be prevented from breaking the boundaries of the DSL (Sec. 2.2.5). Whereas this is can be a clear disadvantage in domains outside of the software engineering domain, the possibility to mix GPL code with MTL code can be considered a valuable advantage by a programmer who is experienced with the host language. With an internal MTL, one can provisionally break the boundaries of the DSL for a specific task which is difficult to accomplish with the limited means of the MTL, without rejecting the MTL completely. It can be reviewed later whether the task can also be accomplished by means of the MTL or whether the MTL can be extended with the required capability. With our approach we want to provide an internal MTL which compares well with existing external MTLs – ATL in particular – while requiring less effort for providing MTL tooling.

### 4.1.3 Scala vs. Other Host Languages

Some GPLs are better suited as host languages than others because they offer more syntactical flexibility and more possibilities to create the desired DSL syntax. We illustrated this with the example of an internal query DSL in Sec. 2.2.5. Languages which are considered as good host languages are Ruby, Smalltalk, Lisp (including dialects), Groovy, and Scala (Fowler, 2010; Günther and Cleenewerck, 2010). Most of these languages provide some sort of an *open class* concept, which means that the perceived behaviour of a built-in type can be modified within the scope of an internal DSL (in Scala this is achieved with implicit conversions as explained in Sec. 2.4.4).

With the exception of Scala, all of these languages are dynamically typed with little or no compile time type checking. However, a commonality of all model transformation applications is the typed nature of transformation sources (and in many cases also transformation targets) by means of metamodels. Apart from defining constraints on how model elements can be combined, a metamodel defines a set of types – e.g., as classes – and each element in a model has at least one of these types. Thus, a metamodel can be considered to constitute the type of a model. Due to the importance of types for model transformations, and because Scala is one of few statically typed languages which are

[4]http://projects.ikv.de/qvt
[5]http://scala-ide.org

considered to be good host languages for internal DSLs, we regard Scala as a natural choice for implementing internal MTLs.

An important reason for favoring a statically typed host language over a dynamically typed one is that tools for dynamically typed languages are usually unable to provide the same level of assistance that tools for statically typed languages provide. The reason is that the tooling of a statically typed language has more information available at editing time that can be used to assist the user. Features like code completion and error highlighting are usually more accurate with tools for statically typed languages. Because reusing the tooling of the host language is the main advantage of the internal DSL approach, a poor level of assistance provided by the host language's tooling would render this advantage useless.

Finally, because Scala is both statically typed and a JVM language, seamless integration with EMF and EMF-specific tool support is possible. Because EMF generates Java classes from metamodels, Scala tools can access those types and provide user assistance based on this information, such as code completion or even debugging. Furthermore, as EMF is implemented in Java, developers working with transformations of EMF-based models are most likely proficient in Java. As Scala's syntax is intentionally close to that of Java, EMF developers are more likely able to take advantage of the possibility to mix Scala code with MTL code.

In Table 4.1 we compare potential host languages for implementing an MTL as an internal DSL. We compare languages by four criteria: (1) 'DSL-ability', i.e., how well the syntax supports custom syntax constructs and extension of existing syntax elements. For instance, Java's syntax is rather rigid and does not allow much modification to be made. (2) Static type system, i.e., how well the language's type system supports to statically check transformations in order to achieve metamodel-awareness. (3) JVM-integration, i.e., how well a language integrates with Java-based frameworks such as EMF. (4) Language popularity and language tool support. Popularity is an important criteria because the user of an internal DSL should have at least basic knowledge of the host language.

| | DSL-ability | Static type system | JVM integration | Popularity & tools |
|---|---|---|---|---|
| Java | + | + | +++ | +++ |
| Python | + | N/A | + (Jython) | ++ |
| Ruby | +++ | N/A | + (JRuby) | ++ |
| Scala | +++ | ++ | ++ | ++ |
| Haskell | ++ | +++ | N/A | ++ |
| Clojure | +++ | N/A | ++ | + |
| Groovy | +++ | + (since 2.0) | ++ | + |
| Xtend | ++ | + (since 2.0) | ++ | + |

Table 4.1: Comparison of potential host languages for internal transformation languages

The choice of host language candidates and their rating is subjective and we do not claim completeness by any means. The chosen languages are ordered from top to bottom

by their popularity[6]. The tool support rating is partly influenced by a language's typing because a statically typed language can usually provide more advanced assistance, for instance, more precise auto-completion suggestions.

Summing up our rationale, any statically typed JVM language which is suited for internal DSLs could serve as a host language for our approach. At the time we started with our work, Scala was simply the only language which met those requirements. By now, both Xtend[7] and Groovy[8] have been extended with a static type system so that they could also be considered good candidates for our approach. Xtend has the advantage that it is tightly integrated with the modelware technologies which we use, particularly with *Xtext*. Therefore, Xtend's popularity in the modelware technological space is higher than its still very low general popularity. Groovy, on the other hand, has recently gained popularity as a host language for internal JVM DSLs. *Gradle*[9] is a very successful Groovy-based internal DSL for build automation. However, also in comparison to Groovy and Xtend as they appear now, Scala still seems like a good choice for our approach. Apart from the highest popularity of the suitable languages, Scala also has the most advanced type system. As we show later, we make extensive use of Scala's type system for achieving metamodel-awareness. This would not be possible with Xtend or Groovy.

One purpose of the following sections is to evaluate Scala as a host language for MTLs, especially in comparison to dynamically typed languages, Ruby in particular. As an example, in the next sections, we present how to implement an internal DSL for unidirectional model-to-model transformations in Scala that is similar to ATL. We compare this language and its tool support with ATL as an example of an external MTL and with RubyTL as an example of an internal MTL hosted in Ruby (Cuadrado et al., 2006).

## 4.2 A Basic ATL-like Transformation Language in Scala

To demonstrate Scala as a host language for MTLs, we develop an internal Scala MTL which is – using terminology from Czarnecki and Helsen, 2010 – *rule-based* and *unidirectional*, and enables the description of hybrid *declarative* and *imperative* model-to-model transformations with a *new target source-target relationship*. This language is designed to resemble ATL. Consequently, we demonstrate the usage of our Scala MTL with the help of ATL's well-known Families2Persons[10] example, and compare the syntax of our Scala MTL with that of ATL.

### 4.2.1 A Simple Transformation

The basic example called Families2Persons from the ATL tutorials is a model-to-model transformation. The Families metamodel is shown in Fig. 4.1. Every family member is to

[6] according to the Redmonk Programming Language Ranking 2014 (http://redmonk.com/sogrady/2014/01/22/language-rankings-1-14/)
[7] http://eclipse.org/xtend/
[8] http://groovy.codehaus.org
[9] http://gradle.org
[10] http://wiki.eclipse.org/ATL/Tutorials_-_Create_a_simple_ATL_transformation

be transformed into a person. A person can either be male or female; a person has only one full name. The Persons metamodel is shown in Fig. 4.2.

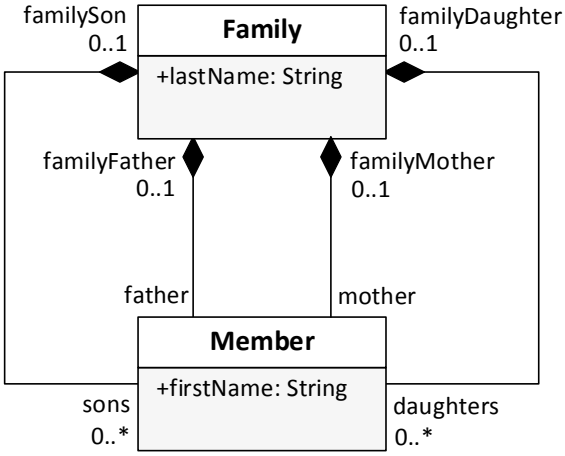



Figure 4.1: Families metamodel

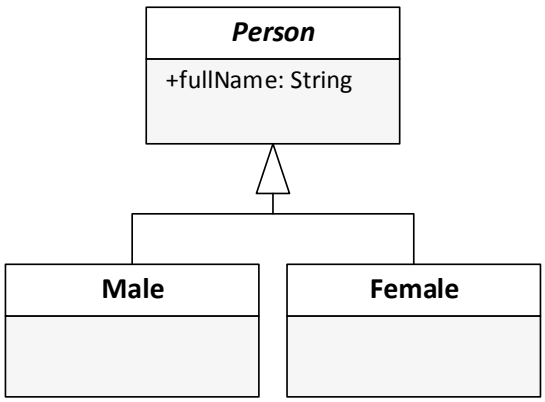



Figure 4.2: Persons metamodel

The transformation creates a person for each member in the source model. The transformation needs to determine the gender of each family member and then creates a new male or female person, respectively. The gender of a family member is determined by the way it is referenced from the family. Finally, the full name is set according to the first name of the member and the last name of the family it belongs to.

### 4.2.2 Rule Definition

ATL transformation rules describe the transformation from a source model element to a target model element. Listing 4.1 shows an ATL rule which transforms a member to a female person. An ATL rule has different sections: two mandatory sections, 'from' and 'to', and two optional sections, 'using' and 'do'. A rule specifies types, i.e., metamodel classes, of its source and its targets within the 'from' and 'to' sections. In this example, a helper method `isFemale` implemented in ATL's imperative function syntax (lines 11–20) is used to check the gender of the source member (line 3). The helper method determines the gender by checking which of a member's four possible back-references to its family is actually set, using the method `oclIsUndefined()`, in other words it checks for non-null fields. This ensures that the rule `Member2Female` is only executed upon members whose non-null back-reference to their family is `familyMother` or `familyDaughter`. Within the 'do' section, the newly created female person is assigned its full name based on the source member's first name and the last name retrieved from the family by another helper method called `familyName` (line 7), whose definition we do not show here.

In our Scala MTL, rules are instances of the class `Rule` and source and target types are specified as type parameters. Listing 4.2 shows the `Member` to `Female` rule in our Scala MTL. The class `Rule` provides methods that – by omitting parenthesis, dots, and semicolons – act as the keywords of our MTL. Lines 1–7 could as well be writ-

Listing 4.1: Rule *MemberToFemale* using ATL

```
rule Member2Female {
  from
    s: Families!Member (s.isFemale())
  to
    t: Persons!Female
  do {
    t.fullName <- s.firstName + ' ' + familyName(s);
  }
}

helper context Families!Member def: isFemale(): Boolean =
  if not self.familyMother.oclIsUndefined() then
    true
  else
    if not self.familyDaughter.oclIsUndefined() then
      true
    else
      false
    endif
  endif;
```

ten as `new Rule[Member,Female]().when(isFemale).perform(...);`. Method chaining follows the fluent interface pattern as explained in Sec. 2.4.3. The methods of class `Rule` are parameterized with functions. The types of these function parameters and the types of their parameters are determined by the rule's type parameters. Due to type inference, these types do not have to be specified again inside the rule. Similar to ATL's 'from' section, a `when` method is used to define execution constraints, which are passed as a function object (line 3). The passed function (defined in line 9) has to take an object of the rule's source type as input and has to return a boolean value.

Listing 4.2: Rule *MemberToFemale* using the Scala MTL

```
new Rule[Member, Female]
  when
    isFemale
  perform
    ((s, t) => {
      t.setFullName(s.getFirstName() + " " + getFamilyName(s))
    })
// using Scala as a GPL for helper methods:
def isFemale(m: Member) = m.familyMother!=null || m.familyDaughter!=null
```

The actual transformation logic is passed as a function object to the `perform` method – 'do' is already a keyword in Scala and cannot be used here. The passed function has to have two input parameters with types which correspond to the rule's source and target types. In the example, this function is defined anonymously (line 5-7) and the types of its parameters `s` and `t` (source and target) are inferred automatically.

Scala's support for function passing and anonymous function definition as well as Scala's flexible syntax which allows dots and parentheses to be omitted in method invocations, are the two main facilities which we use to create the look and feel of an external language. In contrast to other statically typed languages like Java, Scala's type infer-

ence helps to keep the code clean and not cluttered by type annotations. Furthermore, because of Scala's functional programming features and its concise syntax, a separate imperative syntax as provided by ATL can be avoided. Scala also serves as a well-integrated alternative to OCL queries as shown by Křikava and Collet (2012).

### 4.2.3 Transformation Execution

No special tooling or plug-ins are needed for transformation execution. A transformation is an instance of the class `TransformationM2M` which manages transformation execution. It is parameterized using keyword methods with source and target metamodels as shown in Listing 4.3 (line 4-5). One or many rules can be added to the transformation using the `addRule` method (line 7). Calling the `transform` method with the source model as argument starts the transformation. The source model can be provided either as a resource link to a model persisted as an XML file, or as an in-memory object of type `Iterable<EObject>` which provides the model's traversable object graph. Because an EMF model has a spanning containment tree, the model element which is the root node of that containment tree can be used to obtain such a traversable graph of all of a model's elements. The root element therefore acts as a *handle* to the model – we will explain this in more detail, and also what the type of a model is, in Sec. 4.4. To chain transformations, other transformations can be used as argument for the `transform` method as well. Finally, the transformation result is returned as an EMF resource, which is the default behaviour, or can be saved to the file system by calling `export` as shown in line 9.

Listing 4.3: Transformation execution example

```
1 val member2female = new Rule[Member, Female] ... // as in listing 4.2
2 ...
3 val transformation = new TransformationM2M
4   from "http://../Families"
5   to "http://../Persons"
6
7 transformation addRule member2female
8
9 transformation transform sourceModel export "output.xmi"
```

During transformation execution, the source model is traversed. Rules can be marked as 'lazy' if they are not to be applied on source model elements directly. A rule is only executed, if the source type matches and if the `when` function returns true. This is comparable to 'matched' and 'lazy' rules in ATL or to 'top' and 'normal' rules in RubyTL. *Phasing* as presented by Cuadrado and Molina (2009) for RubyTL is also supported by calling the transformation's `nextPhase` method between adding rules.

The transformation process keeps traces, which store the created target model elements and the rules used for their creation. Traces can be queried within a transformation. By default, new target objects are only created, if there is matching trace, in other words, if the rule has not already been applied to that source element. Alternatively, a transformation rule can explicitly be declared to create new elements every time by calling the rule's `isNotUnique` method. This is similar to 'copy rules' in RubyTL.

### 4.2.4 Extending the Language: Multiple Target Model Elements

One of the advantages of internal DSLs is their easy extensibility in contrast to external DSLs where DSL-specific tools have to be adapted accordingly. In this section, we demonstrate with a simple example how to add functionality to the Scala MTL.

In the simple example of Listing 4.2 one object of type `Member` is always transformed into one object of type `Female`. Other transformation languages allow the creation of more than one target object per rule. This can be a list of objects of the same type – often called one-to-many rule – or objects of different types.

Regarding different target types, a drawback of using type parameters to define a rule's source and target type is the fixed number of type parameters. Scala's type system does not allow classes to be overloaded with a different number of type parameters. In order to allow a rule to have more than one target type, we could define different rule classes with a different number of type parameters, such as `Rule2[S,T1,T2]`, `Rule3[S,T1,T2,T3]` etc. However, this leads to duplication of code. To avoid this, one could use *heterogeneously typed lists* (Kiselyov et al., 2004). However, this increases code complexity considerably.

We apply a more lightweight solution. We provide a statically available Scala object which enhances the syntax of the Scala MTL with a `create` method. With this method, we can create additional output objects without changing the rule's 'signature'. The method has a type parameter to determine the target object's type (refer to Listing 4.4, line 4). All attributes and methods of the object created by this `create` method are accessible as usual. However, these additional target objects are not defined within the rule's signature and are created as a side effect. This becomes important when a rule is used implicitly as a function, which we explain in detail in the next section.

Listing 4.4: Creating additional target model elements

```
1 new Rule[Member, Female]
2   perform ((s, t) => {
3     t.setFullName(s.getFirstName() + " " + getFamilyName(s))
4     var newFemale = create[Female]
5     newFemale.setFullName("...")
6   })
```

An obvious alternative to target object creation is the standard `new` operator. However, using `new` is inadequate for two reasons. First, EMF objects should not be created directly but by factories, as we explained in Sec. 2.3.2. Second, new target objects need to be registered in the transformation trace; our `create` method does this automatically.

Furthermore, to enable concise definition of rules with multiple target objects of the same type, the keyword `toMany` can be used instead of `perform`. The `toMany` method expects a function as argument similar to `perform`. But the second argument given to this passed function is a reference to an empty list and not the target object. This list can be filled with an arbitrary number of target objects. However, the same effect as with `toMany` rules can also be achieved with lazy rules.

### 4.2.5 Implicit Rule Application by Static Type-Analysis

Scala's implicit conversions, or shorter *implicits*, greatly improve flexibility, similar to the open class concept in dynamically typed languages, but still provide static type-safety. Implicits are one of the main reasons why Scala is well-suited for building internal DSLs. In this section, we demonstrate how implicits help to implement a declarative, rule-based language.

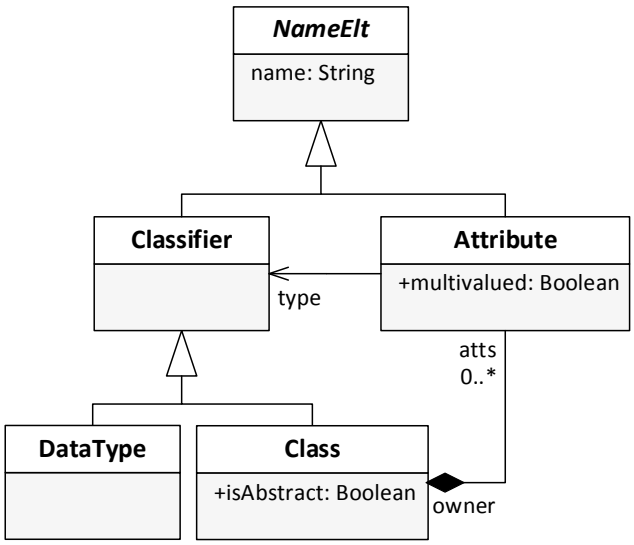



Figure 4.3: Class metamodel

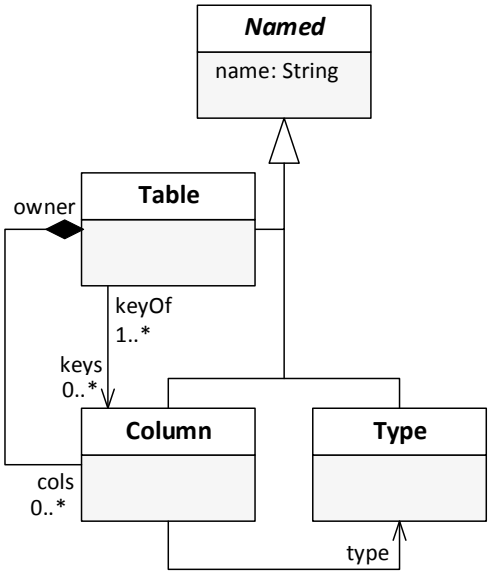



Figure 4.4: Relation metamodel

The following three listings are based on the ATL example Class2Relational[11] where a class schema model (metamodel in Fig. 4.3) is transformed into a relational database model (metamodel in Fig. 4.4). We show how the `Attribute2Column` rule of this transformation can be implemented with ATL, RubyTL, and with our Scala MTL. The rule generates a foreign key column in a table based on an attribute of type `Class`. Additionally, the source attribute has to be a single value and must not be a list (i.e., not *multi-valued*) to trigger the rule.

First, the newly created column gets its name. Second, the value of the column's attribute 'type' is retrieved with a helper method. Finally, the owner of the new column is set to the owning table. This table needs to be the same as the one generated when the owner of the source attribute was transformed.

Our Scala MTL uses implicit conversions to provide a concise syntax: in line 8 of Listing 4.7, a value of type `Class` is passed to the `setOwner` method, which expects a value of type `Table`. Therefore another rule which transforms a class to a table is required. The need for such a rule can be explicitly expressed with our Scala MTL's method `as[ReturnType](inputObject)`. In this example an expression for *explicit* conversion would be `col.setOwner(as[Table](attr.getOwner))`. Similarly, a lazy rule (see

[11] http://www.eclipse.org/m2m/atl/atlTransformations/#Class2Relational

Listing 4.5: Rule *Attribute2Column* using ATL

```
rule classAttribute2Column {
  from
    attr : Class!Attribute (
      attr.type.oclIsKindOf(Class!Class) and not attr.multivalued
    )
  to
    col : Relational!Column (
      name <- attr.name + 'Id',
      type <- thisModule.objectIdType,
      owner <- attr.owner
    )
}
```

Listing 4.6: Rule *Attribute2Column* using RubyTL

```
top_rule 'classAttribute2Column' do
  from Class::Attribute
  to   Relation::Column

  filter do |attr|
    attr.type.kind_of? Class::Class and not attr.multivalued
  end

  mapping do |attr, col|
    col.name = attr.name + 'Id'
    col.type = objectIdType
    col.owner = attr.owner
  end
end
```

Sec. 4.2) is also explicitly called like this. However, the explicit call can be omitted if the required rule was declared to be implicitly available: `implicit val classToTable = new Rule[Class, Table] perform (...)`.

This is possible, because the `Rule` class in the Scala MTL extends the built-in `Function1` Scala type. As a result, a rule can be used like a function with one parameter. The signature of this function is determined by the rule's type arguments. An invocation of the `classToTable` function therefore needs a parameter of type `Class` and returns a `Table`. The Scala compiler inserts invocations of these 'rule functions' automatically to convert objects implicitly as long as the required rules are marked as implicit and are in scope.

In the example, a conversion from the attribute's owner (of type `Class`) to the type

Listing 4.7: Rule *Attribute2Column* using the Scala MTL

```
new Rule[Attribute, Column]
  when ((attr) => {
    attr.getType.isInstanceOf[Class] && !attr.isMultivalued
  })
  perform ((attr, col) => {
    col.setName(attr.getName + "Id")
    col.setType(objectIdType)
    col.setOwner(attr.getOwner)
  })
```

which is needed for the column's owner (of type `Table`) is necessary. The Scala compiler solves this type problem by automatically calling the `classToTable` rule. If no appropriate rule is available, a compile-time error message will report that no suitable conversion could be found or why available conversions did not fit.

This example shows how Scala's type inference and implicits mechanisms can be used to create a syntax that is as concise as in ATL or RubyTL but still preserves static type-safety. In fact, Scala's implicits mechanism is a rule-based system itself and fits in with implementing rule-based transformation languages. However, as the insertion of an implicit conversion is decided at compile-time based on required types, the inserted rule can still fail at runtime because its value-based constraint is not satisfied.

## 4.3 Getting more Declarative: Case Classes & Implicits

In this section, we show how case classes together with implicit conversions can be used to achieve a more declarative syntax. Therefore, in the next subsection, we first demonstrate how a more declarative element creation can be achieved. In the subsequent subsection, we demonstrate how powerful pattern matching abilities can be integrated in the MTL by the use of case classes. Finally, after having demonstrated why it is so useful to implicitly convert between case class objects and EMF model elements, we discuss how those conversions can be generated automatically from analyzing a metamodel.

### 4.3.1 Declarative Element Creation Using Case Classes

Many transformation rules only create a target model element and set its attributes. Therefore, some MTLs provide features to create objects and immediately pass their attribute values along, instead of imperatively using according setter methods (as shown in listing 4.7, lines 6–8). ATL provides a declarative 'to' section (refer to listing 4.5 lines 8–10) as an alternative to the more imperative 'do' section, which we mimic with the 'perform' section in our Scala DSL. In Scala, a similar way for a more declarative object creation is the use of *case classes* (see Sec. 2.4.5). Instances of case classes can be created without `new` and their attribute values can be passed right along.

However, the classes that EMF generates from a metamodel are Java classes and Java does not have the concept of case classes. Furthermore, because EMF does not expose concrete metamodel class implementations directly, but only provides interfaces and factories for element creation, we cannot create model elements directly.

For being able to take advantage of Scala's case classes, we generate a corresponding case class for each class defined in the target metamodel. In addition, we generate an implicit conversion for each case class, so that instances of these generated case classes are converted implicitly to their corresponding target model objects, using EMF's factories in the conversion. The required code can be generated explicitly with a Scala script or with an Eclipse plug-in – we show this after the next subsection. By default, we name a case class like its corresponding metamodel class but with a 'CC' postfix.

Listing 4.8 shows a version of the `Member2Female` rule using a case class for simpler target object creation. Note the use of `FemaleCC` instead of `Female` in line 2. We further

shortened this syntax by overloading the `perform` method with a variant which expects a function with just a single parameter.

Listing 4.8: Object creation using case classes

```
1 new Rule[Member, Female]
2   perform ((s) => FemaleCC(s.getFirstName() + " " + getFamilyName(s)))
```

Here, it is particularly helpful that Scala supports *named* and *default parameters*: In the example above, the target element creation in line 2 could alternatively be written as `FemaleCC(`*`fullName`*` = s.getFirstName() + ...)`. This makes the instantiation of classes with many constructor parameters more manageable.

### 4.3.2 Pattern Matching

Pattern matching is a powerful Scala feature which is particularly useful for transformation code with a lot of alternatives or null checks. To illustrate that, we first show a rule defined in ATL (Listing 4.9). It uses the Relations metamodel (Fig. 4.4). The rule is intended to extract the type of a column whose table's name starts with `"Customer"`. First, null checks for the involved attributes are required (line 7), and then the name of the owning table is tested. However, readability suffers from several nested `If` statements, and a complex pattern structures can easily lead to missing cases. The presented ATL rule, for example, does not cover the case where `type` and `owner` are not null but the `owner name` is incorrect.

Listing 4.9: An ATL rule to select the type of columns in tables called 'Customer...'

```
1  rule ColumnTypeSelect {
2    from
3      c : Relational!Column
4    to
5      type : Relational!Type
6    do {
7      if(not c.type.oclIsUndefined() and not c.owner.oclIsUndefined()) {
8        if(c.owner.name.startsWith('Customer')) {
9          type.name <- c.type.name;
10       }
11     } else {
12       type.name <- 'unknown';
13     }
14   }
15 }
```

With pattern matching, tasks like this are better manageable. However, pattern matching in Scala is only available on instances of case classes because here, the Scala compiler automatically provides the required instance methods. To integrate Scala's pattern matching into our MTL, we generate case classes and corresponding implicit conversions not only for the target metamodel but also for the source metamodel. This way, source model elements are implicitly converted into case class instances, and then patterns can be matched on those instances. Such match relies on the order of constructor parameters of the case class. We implemented this order to be alphabetic by the

attributes' name. For the example above, this results in the following case class constructors: `ColumnCC(keyOf, name, owner, type)`, `TypeCC(name)`, and `TableCC(cols, keys, name)`. Listing 4.10 shows a rule in our Scala MTL which is similar to the ATL rule above and which uses those case class constructors for *deep pattern matching*. This means that the pattern does not only specify attribute values of matched element itself but also of the attribute values of the elements contained in it. In our MTL, pattern matching is made available by using '`use.matching`' instead of the `perform` keyword. This way, the implicit conversion into case class instances is automatically triggered.

Listing 4.10: Pattern matching using generated case classes

```
1 new Rule[Column, Type].use.matching {
2   case ColumnCC(_, _, TableCC(_, _, name), t@TypeCC(_))
3     if name.startsWith("Customer") => t
4   case _ => TypeCC("unknown") // the default case
5 }
```

Scala's pattern matching renders null checks on attributes unnecessary. For instance, the rule shown in Listing 4.10 will not fail if the column's `type` attribute is null. Instead, the default case (line 4) is triggered because a column with a null-valued `type` attribute does not match the pattern specified in line 2; `null` is only matched by `null` (explicit null check) or by '_' which matches on everything. Pattern matching therefore enables fine grained error handling. Each unsatisfying attribute occurrence can be addressed explicitly with a case statement. This allows the effective separation of error handling code and actual logic triggered by the desired input pattern.

Because of the way pattern matching is currently implemented in Scala, pattern matching code can become cluttered with occurrences of '_' when a class has many attributes but only some of them are matched for. This is going to be improved when Scala's support for named parameters is not limited to case class creation anymore but is extended to case class matching. This is currently scheduled[12] for Scala version 2.12.

### 4.3.3 Case Class Generation and Conversion

In the previous two subsections, we demonstrated that it is useful to implicitly convert between EMF model elements and case class objects, in order to achieve a higher expressiveness of our Scala MTL. In this section, we first explain how the required case class definitions as well as the corresponding conversion functions can be generated automatically from metamodels. Afterwards, we explain in more detail how the runtime conversion between model elements and case class objects is performed.

#### Generating Case Class Definitions and Conversions

To convert an EMF model element into a Scala case class instance, a corresponding case class definition has to exist. One could provide those definitions manually but this would be tedious and error-prone. We therefore generate case class definitions automatically from the metamodel. For each metamodel class, one case class definition and two implicit

[12] https://issues.scala-lang.org/browse/SI-5323

conversions need to be provided: one function which converts an EMF model element (an instance of a metamodel class) into a corresponding case class instance, and one function which converts a case class object into an EMF model element. In contrast to their corresponding metamodel classes, the case classes we generate do not define any methods, but only define public attributes – only those are needed for pattern matching and element creation. Because case classes cannot be inherited, we need to implement inheritance between metamodel classes by defining inherited attributes explicitly in every case class. In other words, the generated case classes merely define the data structures of their corresponding metamodel classes. Every generated case class implements the interface `IGeneratedCC` which is used to tag a class as being generated. This way, in the implementation of our MTL, methods working with case class instances can require this type and can abstract over concrete case class types, whose definitions may not be generated yet. Figure 4.5 shows how an inheritance hierarchy in the metamodel is translated into corresponding case class definitions.

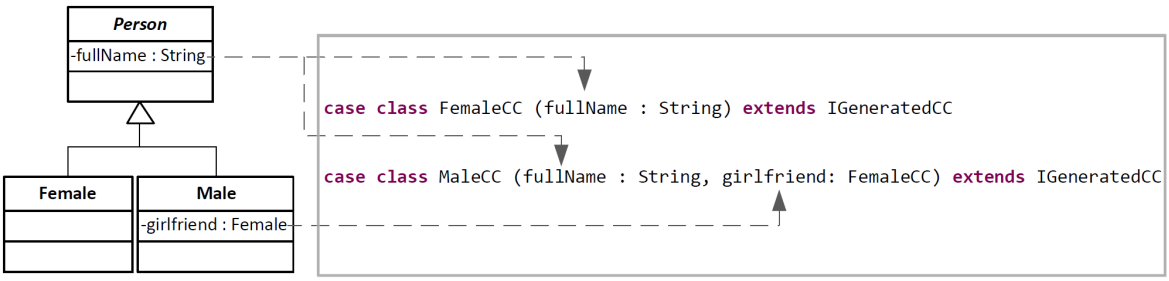



Figure 4.5: Generating case class definitions from a metamodel

Because an EMF metamodel is an EMF model, too, we implement the case class generation and the implicit conversion generation as a model-to-text transformation by using our MTL itself. This way, the case class generation is a standard part of our MTL so that the MTL is self-contained, and only a Java runtime is needed to generate case classes and conversions. The case class generation transformation also allows for circular dependencies between metamodel classes. In addition to the transformation itself, we also provide a plug-in for *Eclipse*, where a metamodel can be selected and the corresponding case class generation can be triggered. However, using this plug-in is optional as it creates a dependency of the internal DSL to one specific Scala tooling. Figure 4.6 shows how case class generation can be selected as a run configuration in *Eclipse*, if the plug-in is installed.

### Converting Models to Case Class Objects

When a corresponding case class is defined for each metamodel class, converting from a model element to the corresponding case class object seems straight forward: A conversion of an object of type `Female` from the Families2Persons example could be defined as: `implicit def female2femaleCC(f:Female) = FemaleCC(f.fullName)`. However, as models are graphs which may have cross-references, circular dependencies can

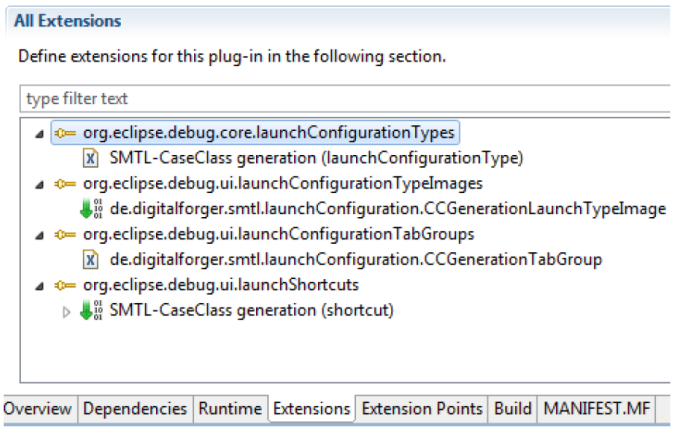



Figure 4.6: An *Eclipse* plug-in provides a run configuration for case class generation

occur. Implementing those conversions naively can therefore easily lead to the following problems: Either the compile-process will not terminate because it does not stop to insert calls to conversion functions, or the conversion-process at runtime will not terminate. Therefore, we apply the following approach to implement the implicit conversions: The first time an implicit conversion is requested, the whole model – that is, the in-memory graph of EMF-based class instances – is converted to a graph of corresponding case class objects, which is then made available globally, so that subsequent conversion requests can simply return an already existing case class instance of that graph.

The algorithm, illustrated in Fig. 4.7, is as follows: (1) A reference to the model element whose conversion is requested is saved in the type-specific conversion function. (2) A general conversion function is called, which first uses EMF's containment hierarchy to identify the root of that containment tree by walking up the hierarchy from the given element. (3) The containment tree is traversed for a first time and – by using the type-specific implicit conversion functions that we generated beforehand from the metamodel – for each model element a corresponding case class object is created and its attribute values are set accordingly. However, fields holding a cross-reference are only set with a type-safe placeholder because the corresponding case class instance of the referenced model element might not exist yet. Importantly, forward traces are kept globally, documenting which case class instance was created from which model element. (4) The model containment tree is traversed for a second time and – by looking up already created case class instances in the traces – placeholders in case class instances are resolved and cross-references in the case class graph are set. (5) The general conversion functions returns to the type-specific conversion function, which uses the created traces to return the corresponding case class instance of the originally passed model element.

After this conversion process was triggered once, every time an implicit conversion of a model element into a case class instance is requested, the conversion function only looks up the corresponding – already created – case class instance and returns it (i.e., only step 5). This way, circular dependencies can be handled. Furthermore, repeatedly converting the same model element in different transformation rules is prevented effectively.

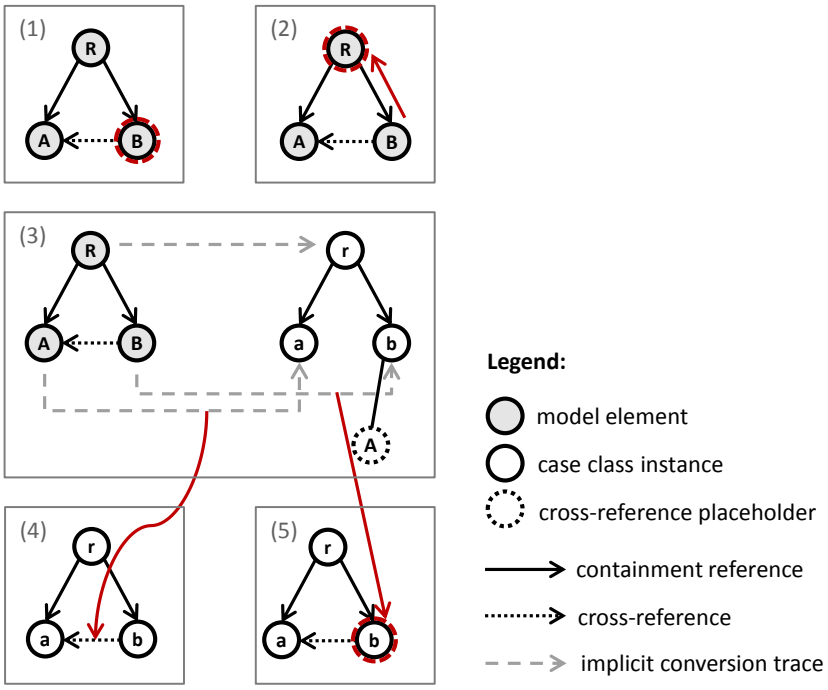



Figure 4.7: Generating a graph of case classes using conversion traces

## 4.4 Towards More Metamodel-Awareness: What is a Type of a Model?

We want to use Scala's type checking to provide metamodel-aware tooling. However, our MTL's tooling is only metamodel-aware insofar as it is type-checked whether a single model element is an instance of a given metamodel class and therefore adheres to the constraints specified by that class. So far, we cannot type-check which class belongs to what metamodel. If we consider a metamodel to constitute the type of a model, then the question is: how can we encode a metamodel in Scala's type system in such a way that it can be type-checked whether a model conforms to a given metamodel?

Of course, the goal of using Scala's type checker to check models against metamodels cannot be completely achieved because a metamodel can contain OCL constraints which refer to runtime values. We cannot type-check those constraints statically at compile-time. However, in the following subsections, we show that we can at least check whether a transformation rule's input element's class and the output element's class belong to the transformation's source and target metamodel, respectively.

### 4.4.1 A Type System Representation for Metamodels

A modelware metamodel contains a set of classes and, optionally, a set of additional constraints. As classes define types, a metamodel defines a set of types. To type-check a model we have to check its elements against this set of types. Programming languages

such as Java, Scala, or C# provide packages as a means to define a set types. However, packages are only used for scoping, visibility, and deployment. It cannot be type-checked whether a given type belongs to a certain package. For the type system, the only way to check whether a type belongs to a certain set of types is by subtyping relations, that is, whether a type belongs to the set of supertypes or subtypes of another type. We therefore define a set of types by a common supertype. A common strategy for this is to define a 'tag interface' which does not define any methods but can be used to tag another type. Because multiple inheritance is allowed, any type can be tagged and also one type can be tagged by different interfaces. Thus, type sets defined this way can overlap. This is important because a class can belong to multiple metamodels.

However, the generated Java classes which EMF generates from a metamodel are not tagged that way. One could modify the way EMF generates classes or could modify the generated classes afterwards. However, this would risk the compatibility with other EMF-based technologies, and we want our Scala MTL to integrate well with existing those technologies. Instead, we use the case classes definitions which we generate from a metamodel. Such code generation can easily be adjusted so that every generated case class implements an interface which is associated with the metamodel to which the corresponding metamodel class belongs.

### 4.4.2 The Runtime Representation of a Model

In section 4.2.3, we mentioned that in our Scala MTL, a transformation's `transform` method expects either a resource link to an XMI file or an in-memory object of type `Iterable<EObject>`. We can now define more precisely what a model in our MTL technically is. At runtime, a model is a graph of EMF objects – i.e., instances of classes that inherit from `EObject` – which are all part of the same EMF containment hierarchy, which means they can be traversed using `EObject.getChildren()`. Therefore, the root object of the containment hierarchy can act as a 'handle' to the model, from which we can obtain all elements of the model. The type of this specific root object is defined by its metamodel class. One must therefore distinguish between the type of the model, being defined by the metamodel, and the type of the handle of the model, being defined by one specific metamodel class. The type of the model is considered the set of types which are defined by a metamodel's classes, or – as proxies for those types – the types defined by the corresponding generated case classes, which are all subtypes of the metamodel's tag interface. If all objects in a model's graph of objects are instances of these metamodel classes or case classes, we say that a model conforms to the type constituted by its metamodel.

### 4.4.3 Defining Type-Safe, Metamodel-Aware Transformations

We can now change our MTL's class `TransformationM2M` so that a transformation's source and target metamodel are specified as type parameters. With appropriate type constraints for the type parameters of the `addRule` methods, the type-checker can now ensure that all rules added to a transformation actually work on the specified source

and target metamodel. As we generate case classes from the metamodel anyway, we also generate a type which represents the metamodel. This type inherits from trait `Metamodel` – a *trait* is Scala's counterpart of an interface in Java – and contains the corresponding tag interface. Listing 4.11 shows parts of the modified class definitions in our Scala MTL, an example of code generated from a metamodel, and how the modified MTL methods are used in order to define metamodel-aware transformations.

Listing 4.11: Type-safe, metamodel-aware transformation definition

```
1  // Defined by our Scala MTL:
2  trait Metamodel { type Tag; type Root <: Tag with IGeneratedCC; val uri: String }
3  class Rule[Source, Target] { ... }
4  class TransformationM2M[SrcMM <: Metamodel, TrgtMM <: Metamodel] {
5    def addRule[Src <% SrcMM#Tag, Trgt <% TrgtMM#Tag](rule: Rule[Src,Trgt])
6    def transform[SrcModel <% SrcMM#Root](srcModel: SrcModel): TrgtMM#Root
7    ...
8  }
9  // Code generated from metamodel Families:
10 object FamiliesMM extends Metamodel {
11   trait FamiliesTag
12   type Tag = FamiliesTag
13   type Root = FamilyCC
14   val uri = "http://..."
15   case class FamilyCC(...) extends FamiliesTag with IGeneratedCC
16   case class MemberCC(...) extends FamiliesTag with IGeneratedCC
17   ...
18 }
19 // Using our Scala MTL for metamodel-aware transformation definition:
20 val fams2pers = new TransformationM2M[FamiliesMM, PersonsMM] // no from/to anymore
21 val member2female = new Rule[FamiliesMM.MemberCC, PersonsMM.FemaleCC] when .. perform ..
22 fams2pers addRule member2female // no type params needed because of type inference
23 fams2pers addRule ...
24 val targetModel = fams2pers transform sourceModel
```

In line 2 the general structure of a type which represents a metamodel is defined. It contains a tag interface `Tag`, the type `Root` of a model's 'handle', and a universal resource identifier (URI). Class `Rule` is not changed and therefore only sketched in line 3. Class `TransformationM2M` now has two type parameters for specifying the source and the target metamodel. Thus, type arguments are constrained to be subtypes of the metamodel trait defined in line 2. Now, the constraints of the type parameters of the `addRule` method ensure statically that a rule added to the transformation matches with the specified metamodels: `Src <% SrcMM#Tag` specifies that the source element's type of the added rule must be *viewable* as a type which implements the tag interface of the transformation's source metamodel. This means that either the type itself must implement the tag or there must be a suitable implicit conversion with such output type in scope. That way, when defining a new rule, one can either specify the types of the metamodel elements or their corresponding case class types. Similarly, in line 6 the `transform` method ensures statically that the passed object conforms to the root type specified in the metamodel.

Lines 10–16 sketch which code is generated from a metamodel – in this case from the Families metamodel. A tag trait is defined, the root type is specified, and the generated case classes implement both the tag interface and the `IGeneratedCC` interface which all

generated case classes implement.

As a result, in line 20, a transformation such as Families2Persons is now created with type arguments specifying the source and target metamodel, instead of just specifying the metamodel URI using the 'from' and 'to' methods as shown in Listing 4.3 on p. 86. When defining a rule such as *member2female* in line 21, source and target element types are now specified with explicit reference to their metamodel. The syntax of adding rules and transformation execution has not changed. The Scala type-checker is capable of inferring type arguments (Sec. 2.4.6), so that they do not have to be specified explicitly, and statically checks the inferred type arguments against the specified type parameter constraints.

That way, we can at least use Scala's type checker to guarantee that all model elements conform to the metamodel classes of the specified metamodel. Beyond that, some constraints can also be encoded rather easily in the type system, e.g., whether an attribute is multi-valued or not by checking if it is of a subtype of `List`. Other metamodel constraints are more difficult to encode in such a way that Scala's type checker can statically guarantee them, say, a specific cardinality of six. However, we will show in the next chapter, that we can even encode such constraints in the type system, although with more effort.

## 4.5 Related Work and Discussion

In this section, we discuss advantages and disadvantages of our approach and compare it to existing approaches. After presenting related work in the next subsection, we discuss tool support and EMF integration in the subsection thereafter. In the subsequent subsection, we compare the expressiveness of our MTL with existing MTLs by the help of code complexity metrics. Afterwards, we conclude this chapter.

### 4.5.1 Related Work

The general idea and best practices of internal and external DSLs have been extensively discussed by Fowler (2010) on his blog which was later edited into a book. A set of patterns for internal DSL development in several host languages has been published by Günther and Cleenewerck (2010). Scala's potential as a host language for general DSLs has been evaluated in Pointner (2010). Hofer et al. (2008) showed the extensibility of DSLs written in Scala. Scala has already been used as a host language for a variety of internal DSLs, e.g., by Spiewak and Zhao (2010) and by Barringer and Havelund (2011). Sloane (2008) showed how the term-based transformation language *Stratego*[13] can be implemented as an internal DSL in Scala.

Picard (2008) showed how to use Scala for EMF model transformations. However, no domain-specific model transformation constructs or syntax elements were implemented. The work basically shows how to parse an EMF model from its XMI serialization, create Scala objects from it, and how Scala as a GPL can be used to implement transformations. Therefore, the fact that Scala is JVM-based is not leveraged and there is no integration

[13] http://strategoxt.org/

with EMF-based tooling. In the context of MDE, Křikava and Collet (2012) showed how an internal Scala DSL can serve as a powerful alternative to OCL.

Cuadrado and Molina (2006) use Ruby as host language for their MTL called RubyTL. Similar to the transformation language presented in this chapter, they designed RubyTL to be a hybrid transformation language which uses declarative constructs to realize pattern matching and rule selection and an imperative style to realize rule actions. Furthermore, RubyTL is designed as an extendable MTL with the goal to efficiently implement and evaluate new transformation techniques. In Cuadrado and Molina (2008, 2009), the authors facilitated these characteristics to research rule factorization and composition techniques based on rule phasing. Ruby is a dynamically typed language with fast prototyping capabilities but also a lack of static type checking. This stands in direct contrast to our work with Scala. Scala uses static type inference, which enables a similar programming style as in dynamically typed languages. Furthermore, Scala as a host language provides better tool support than RubyTL (or any other dynamically typed language) due to the annotation of static errors and superior content assist based on editing-time knowledge about the type of a variable. Furthermore, RubyTL is not based on EMF or a comparable modeling framework, and works directly on an in-memory representation of XMI. RubyTL therefore does not use any metamodel information and even if Ruby was statically typed, RubyTL would have no types to work with. Additionally, most existing modeling APIs are written in Java and as such can be used from within Scala, but cannot (at least not without limitations) be used in Ruby.

### 4.5.2 Tool Support & EMF Integration

A main rationale for implementing an MTL as an internal Scala DSLs is the low-effort tool support and integration with existing modeling technologies. Basic tool support is 'for free' for internal DSLs since the host language's tools can be used. For external DSLs like ATL, specific tools need to be developed and their quality depends directly on the effort put into them. Compared to other internal DSLs like RubyTL, our approach allows better code completion and error detection based on compile-time type information. Because of Scala's Java interoperability, Java-based modeling frameworks such as EMF or *Kermeta*[14] can be used effortlessly. By accessing Java classes, content assist is provided for metamodel elements, as long as there are corresponding Java classes (as is the case with EMF).

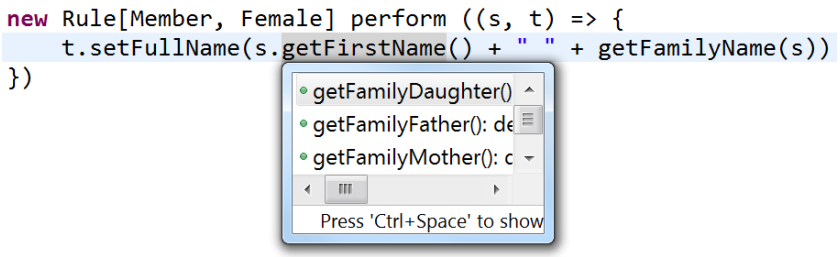



Figure 4.8: Code completion for our Scala MTL using *Eclipse* with the Scala IDE plug-in

[14]http://kermeta.org

To use the presented approach, a Scala compiler and EMF is required. We used the *Scala IDE*[15] plug-in for *Eclipse*, which includes a Scala compiler, to get the described tool support. This tooling provides syntax highlighting, wizards, templates, debugging and code completion. Noteworthy, the transformation code can be debugged like any other Scala program and all attribute values can be observed at runtime, includes those of EMF model elements. Within the listings in this chapter we highlighted the 'keywords' of the internal DSL, although this would not be the case in an unmodified Scala tooling, but could be provided by a separate plug-in.

A dynamically typed language (such as Ruby) allows only limited code completion. Therefore, RubyTL for example offers an *Eclipse* plug-in called AGE[16]. It provides a Ruby editor with syntax highlighting and code templates for RubyTL. The editor's code completion is limited to the keywords of RubyTL since no static type information is available. Errors based on wrong types can only be discovered at runtime. For ATL, an external DSL, a specific rich-featured editor has been developed. In return, ATL's syntax could be perfectly tailored. However, ATL uses only a small set of data types[17]. Therefore, full support in the editor can only be offered for those types. Others will be presented as a default data type named `OclAny`.

### 4.5.3 Expressiveness

For evaluating the expressiveness of the unidirectional MTL which we developed in this chapter, George (2012) calculated code complexity metrics and compared them with existing MTLs, both external and internal. Note that with 'expressiveness', we mean that much can be expressed with little effort. Sometimes expressiveness is only defined as how much can be expressed at all. However, if interpreted this way, a low-level language like assembler would be most expressive because one can basically express every possible computation, just not very concisely.

For measuring the complexity of model transformation descriptions, we present the different components of the Halstead (1977) metrics as well as, for completeness, the lines of code without empty lines, comments, and import declarations (SLOC). Lower numbers represent a better expressiveness. The MTLs which we chose for comparison are ATL and QVT Operational (QVTo, a unidirectional MTL defined in the QVT standard) as examples of external MTLs, and RubyTL as an example of an internal MTL. Results which are significantly better than the average in this comparison are highlighted green, results which are significantly worse, are highlighted red. Table 4.2 shows the results from the simple FamiliesToPersons example. Table 4.3 shows the results from the slightly more complex ClassToRelation example. In these examples our Scala MTL compares well with existing MTLs. It is in most cases more expressive than QVTo, sometimes even better than ATL and similar in complexity to RubyTL. However, in comparison with RubyTL tool support is more advanced because of Scala's static type-safety and EMF integration. In comparison with ATL the effort to provide adequate tooling is reduced significantly.

[15] http://scala-ide.org/
[16] http://gts.inf.um.es/trac/age
[17] http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language#Data_types

| | Scala MTL | ATL | QVTo | RubyTL |
|---|---|---|---|---|
| SLOC | 27 | 40 | 30 | 42 |
| operators n1 | 19 | 24 | 25 | 20 |
| operators N1 | 119 | 129 | 121 | 92 |
| operands n2 | 24 | 23 | 21 | 23 |
| operands N2 | 69 | 61 | 64 | 72 |
| volume | 1020 | 1055 | 1021 | 889 |
| difficulty | 27 | 32 | 38 | 31 |
| effort | 27540 | 33760 | 38798 | 27559 |
| impl. time (s) | 1530 | 1875 | 2155 | 1531 |

Table 4.2: Comparison results from the FamiliesToPersons example (from George, 2012)

| | Scala MTL | ATL | QVTo | RubyTL |
|---|---|---|---|---|
| SLOC | 61 | 87 | 61 | 80 |
| operators n1 | 20 | 22 | 25 | 20 |
| operators N1 | 254 | 214 | 354 | 216 |
| operands n2 | 45 | 41 | 37 | 39 |
| operands N2 | 210 | 187 | 210 | 204 |
| volume | 2794 | 2396 | 3358 | 2470 |
| difficulty | 46 | 50 | 70 | 52 |
| effort | 128524 | 119800 | 235060 | 128440 |
| impl. time (s) | 7140 | 6655 | 13058 | 7135 |

Table 4.3: Comparison results from the ClassToRelation example (from George, 2012)

### 4.5.4 Conclusions

In this chapter, we used Scala to implement a unidirectional model transformation language as an internal DSL. We showed that Scala can be used as a host language for model transformation languages and is flexible enough to create a concrete DSL syntax which resembles that of existing MTLs such as ATL. Since we use an internal DSL approach, our Scala MTL can be extended: language features can be added and existing behaviour can be adopted to the specific needs of one's current transformation task. Furthermore, Scala is rooted in the Java platform and existing modeling frameworks which are mostly written in Java – e.g., EMF and anything written for it – can be used immediately. Furthermore, transformations or helper methods which were already written in Java can be reused, integrated and gradually migrated to the more concise means provided by the internal DSL. Compared to existing internal MTLs, Scala is statically typed and uses type inference: it provides a clean syntax similar to dynamically typed languages but still provides the benefits of static type checking. These benefits include compile type warnings and errors as well as better code completion based on type information. Compared to external MTLs, powerful tool support including full debugging already exists.

However, our approach also shares the general disadvantages of internal DSLs. In contrast to external DSLs, code completion and error messages are not tailored for the DSL.

Moors et al. (2012) recently proposed an extension to Scala that would allow tailoring of error messages of internal Scala DSLs and therefore would make Scala even more suited for internal DSLs. Nevertheless, some knowledge of the host language is required when using an internal DSL. Internal DSLs are easier extensible than external DSLs because no DSL-specific tools have to be adapted. However, often, advanced features of the host language are used in order to achieve a desired DSL syntax. This can make a DSL's implementation difficult to understand and extensions to the DSL less straightforward. Finally, the ability to mix MTL constructs with GPL code is also a disadvantage because arbitrary GPL code significantly limits possibilities for formal reasoning.

To alleviate some of the disadvantages of an internal DSL, IDE plug-ins could be provided to improve error messages or to provide templates and syntax highlighting for the internal DSL. However, this would eliminate the advantage of being independent from DSL-specific tools and their development.

# 5 A Compositional Language for Bidirectional Model Transformation

In the previous chapter, we developed a unidirectional model transformation language (MTL) as an internal DSL in Scala and discussed the general approach of implementing MTLs as internal Scala DSLs. In this chapter, we apply this approach to the development of a bidirectional MTL.

Providing a bidirectional MTL is particularly important for realizing multi-view domain-specific workbenches because there is a lack of bidirectional MTLs which integrate with modelware tools and support non-bijective synchronizations. Our approach to this is to adapt an existing transformation language which supports non-bijective synchronizations so that it integrates well with modelware tools. As we explained in Chap. 3, to implement view synchronization in a domain-specific workbench such as the *NanoWorkbench*, at least synchronization type $(\frac{1}{2}0\frac{1}{2})^{-}_{\not\Delta}$ has to be supported. We therefore chose to adapt *Focal* presented by Foster et al. (2005), a compositional informationally asymmetric bidirectional *tree* transformation language designed for type $(10\frac{1}{2})_{\not\Delta}$, as this includes support for type $(\frac{1}{2}0\frac{1}{2})^{-}_{\not\Delta}$ when restricting backwards update propagation.

We could have also chosen *GRoundTram* presented by Hidaka et al. (2011), a *graph* transformation language designed for type $(10\frac{1}{2})_{\not\Delta}$ which therefore is even better suited for model transformations, that is, transformation of graphs. However, *GRoundTram* relies heavily on recursive functions to traverse the graph. This is generally difficult to combine with meaningful static type checking (Lämmel and Jones, 2005). Moreover, it is particularly difficult to combine this with our Scala-based approach because in contrast to Haskell, for instance, Scala applies *local type inference*, which means that a recursive function always needs an explicit type annotation. In *Focal*, in contrast, recursion is mainly used for list iteration, which can be avoided in Scala by using standard homogeneously typed collections.

This chapter is structured as follows: In the next section, we introduce *state-based lenses*, the computational framework behind *Focal*, as well as *delta-based lenses*, a useful generalization of state-based lenses. In Sec. 5.2, we present a data model which allows us to apply *Focal* to a modelware setting. Based on this data model, in Sec 5.3 we implement *Focal* as an internal Scala DSL which performs extensive static type checking. In Sec. 5.4, we show how this language can be used for model transformations and how it can be adapted for being able to handle non-containment references. Sec. 5.5 presents related work and concludes the chapter. This chapter is partly based on material which has been published in Wider (2012), Wider (2011), and Wider (2014).

## 5.1 Lenses: A Compositional Approach to Bidirectional Transformations

When synchronizing models bidirectionally two functions of update propagation must be consistent with each other in the sense that they satisfy some invertibility property, say $P$, which is often described in terms of equational *well-behavedness laws* (shown on the next page). With a simple synchronization description, it is often easy to show manually whether such laws hold. However, this can get very difficult with complex synchronization descriptions.

The strength of lenses is their compositional notion: complex synchronizations are composed out of small and well-understood synchronizations (for which it is easy to prove well-behavedness laws) by using a set of combinators that guarantee to preserve the invertibility properties of the sublenses for the composed lens. This enables compositional reasoning, which means that the well-behavedness laws only have to be proved for atomic lenses and for combinators, but not for composed lenses anymore.

Lenses were originally developed for an informationally asymmetric setting. This makes invertibility and combinator-design easier, although it is a strong restriction. Later, different approaches were made for informationally symmetric lenses (Hofmann et al., 2011; Diskin et al., 2011a). However, as informational symmetry makes combinator-design more challenging and is not necessarily required for our multi-view workbench scenario, we focus on asymmetric lenses. The next subsection presents state-based lenses and *Focal*. Afterwards, we shortly introduce the concept of delta-based lenses.

### 5.1.1 State-Based Lenses & Focal

Asymmetric state-based lenses, the computational framework behind *Focal* is restricted to informational asymmetry, i.e., one of the two structures which are synchronized has to be an abstraction of the other. The setting is inspired by the *view-update problem* known in the database community, where a database view – the abstraction – has to be updated when the database changes and vice versa. Given a set $C$ of concrete structures and a set $A$ of abstract structures, a lens comprises two functions:

$$\begin{array}{lrcl} get: & C & \to & A \\ put: & A \times C & \to & C \end{array}$$

The forward transformation *get* derives an abstract structure from a given concrete structure. The backward transformation *put* takes an updated abstract structure and the original concrete structure to yield an updated concrete structure. To allow initial creation of a concrete structure from an abstract one, sometimes an alternative non-incremental backward transformation $create: A \to C$ is added, which uses default values for private parts of the concrete structure. It can be omitted if not needed in the given scenario, i.e., if a concrete structure is never created afresh from an abstract structure. A lens which supports the *create* case needs to be provided with a *default structure* $d \in C$. Fig. 5.1 visualizes the way the lens functions are used to derive an abstract view from a concrete source and how an updated source is constructed when the view changes.

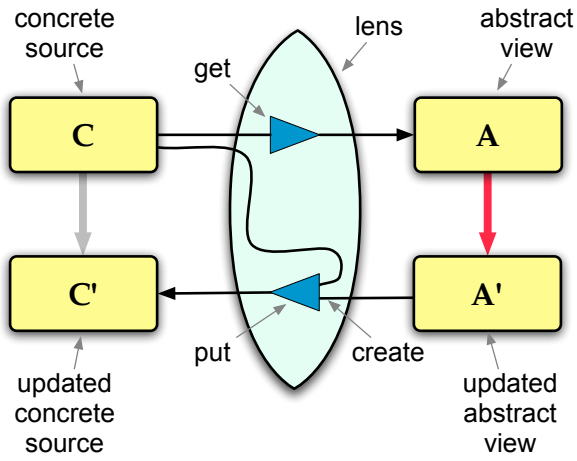



Figure 5.1: A lens synchronizes a concrete source and an abstract view (from Foster, 2009)

Lenses specify *well-behaved* bidirectional transformations, which means that every lens must obey the following *lens laws* (Foster et al., 2005):

$$get(put(a, c)) = a \qquad \text{(PutGet)}$$
$$get(create(a)) = a \qquad \text{(CreateGet)}$$
$$put(get(c), c) = c \qquad \text{(GetPut)}$$

These laws formalize some behaviour one would generally expect from a bidirectional transformation: The updated (or initially created) concrete structure $c$ fully reflects changes made in the abstract structure $a$ (PutGet and CreateGet), and data in the concrete structure that is hidden by the abstraction is preserved (GetPut). In the community of bidirectional transformations, these laws are frequently subject of discussion. Some approaches add more laws or weaken some of those laws presented here.

### General Lenses

The simplest lens is the *id* lens. It does not apply any actual synchronization logic, but only copies whatever it gets to the other side. Because the put function's result is independent from the original concrete structure, which is simply discarded, *id* is a so-called *oblivious* lens. The following listing shows the complete definition of the *id* lens; we use a post-colon notation for type annotation in lens definitions, similar to Scala or UML; we omit (for now) type annotations of the lens functions' parameters as they always refer to the abstract or concrete structure (but are sometimes subject to constraints):

$$\begin{aligned} &id : lens \ \{ \\ &\quad get(c) \quad = c \\ &\quad put(a, c) \quad = a \\ &\quad create(a) \quad = a \\ &\} \end{aligned}$$

It is easy to show that with the *id* lens the lens laws hold. We only need to insert the definitions of *id*'s *put* function and *id*'s *get* function into the PUTGET law:

$$
\begin{aligned}
get(put(a, c)) &= a && \text{(PUTGET law)} \\
get(a) &= a && \text{(id.put inserted)} \\
a &= a && \text{(id.get inserted)}
\end{aligned}
$$

Now, the strength of lenses is their compositional notion: A set of *atomic lenses* – like *id* – whose well-behavedness has been manually proven, is provided together with a set of *lens combinators* for which it has been proven that the resulting composed lens is well-behaved if all of its sublenses are well-behaved. These lenses and combinators can then be used as a vocabulary for bidirectional transformations from which arbitrarily complex lenses can be composed without having to prove the lens laws again.

The most common combinator is the sequential composition *comp* which takes two lenses $l$ and $k$ – its sublenses – as arguments and puts them in a sequence:

$$
\begin{aligned}
&comp(l : lens, k : lens) : lens\ \{ \\
&\quad get(c) \quad\ = k.get(l.get(c)) \\
&\quad put(a, c) \ = l.put(k.put(a, l.get(c)), c) \\
&\quad create(a) \ = l.create(k.create(a)) \\
&\}
\end{aligned}
$$

The *get* function is straightforward: $l$'s *get* function is called and the result is used as input for $k$'s *get* function. The *put* direction is slightly more complicated: first, the original concrete input has to be abstracted by $l$'s *get* function to be a valid input for $k$'s *put* function. As can be seen, a combinator such as the sequential composition is a lens itself, differing from atomic lenses only because it is parameterized with sublenses. A lens created from two lenses using *comp*, is a well-behaved lens – that is, the lens laws hold – as long as the two sublenses are well-behaved (Foster et al., 2007).

### Focal: A Lens Library for Tree Transformations

*Focal* is a lens library based on state-based lenses, that provides a set of atomic lenses and lens combinators for tree transformations. Lenses provided by *Focal* work on *edge-labeled trees* where a tree $t$ is defined as an unordered, potentially empty set of labels which refer to a tree (denoted by $\mapsto$). Fig. 5.2 shows an example of a contact list tree where names refer to a phone number and a URL, encoded as an edge-labeled tree in a horizontal notation.

$$
\left\{
\begin{array}{l}
\texttt{Pat} \mapsto \left\{
\begin{array}{l}
\texttt{Phone} \mapsto \left\{ \texttt{333-4444} \mapsto \{\} \right\} \\
\texttt{URL} \mapsto \left\{ \texttt{http://pat.com} \mapsto \{\} \right\}
\end{array}
\right\} \\
\texttt{Chris} \mapsto \left\{
\begin{array}{l}
\texttt{Phone} \mapsto \left\{ \texttt{888-9999} \mapsto \{\} \right\} \\
\texttt{URL} \mapsto \left\{ \texttt{http://chris.net} \mapsto \{\} \right\}
\end{array}
\right\}
\end{array}
\right\}
$$

Figure 5.2: An example of an edge-labeled tree

As can be seen, all data is encoded as labels (i.e., strings), and leafs are encoded as labels that refer to an empty set of labels (denoted by {}). Now, additionally to general lenses like *id* and *comp*, which are also part of *Focal*, tree-specific lenses are provided. An example of a lens which performs a simple structural tree modification is *hoist*. It is defined as follows:

$$
\begin{aligned}
&\mathit{hoist}(n : \mathit{label}) : \mathit{lens}\ \{\\
&\quad \mathit{get}(c) \quad = t \quad \mathit{if}\ c = \{n \mapsto t\}\\
&\quad \mathit{put}(a, c) \quad = \{n \mapsto a\}\\
&\quad \mathit{create}(a) \quad = \{n \mapsto a\}\\
&\}
\end{aligned}
$$

The *hoist* lens is parameterized with an edge-label $n$. A concrete tree $c$ given to *hoist*'s *get* function must have exactly one root edge which must have the specified label $n$ and leads to $c$'s single child tree $t$. This is the concrete-side constraint of the *hoist* lens. Function *get* then yields this child tree $t$. Thus, *hoist*'s forward transformation removes $c$'s single root edge and thereby flattens the tree by one level. Correspondingly, the two backward transformations *put* and *create* restore the specified root edge by adding it to the potentially modified abstract tree $a$.

A tree-specific lens-combinator which is frequently used in *Focal* to compose more complex tree-specific lenses is *fork*. It splits the given tree by dividing the set of labels into two sets depending on whether a label satisfies a condition $p$ (which is a parameter of *fork*) and then applies one of two lenses (which are the other parameters) for each subtree. Afterwards, the two resulting trees are combined. In the following definition, we denote tree combination – i.e., concatenating the child lists of two trees – by a triple-colon ':::'.

$$
\begin{aligned}
&\mathit{fork}(p : \mathit{condition},\ l : \mathit{lens},\ r : \mathit{lens}) : \mathit{lens}\ \{\\
&\quad \mathit{get}(c) \quad = l.\mathit{get}(\{x{\in}c \mid p(x)\}) ::: r.\mathit{get}(\{x{\in}c \mid \neg p(x)\})\\
&\quad \mathit{put}(a, c) \quad = l.\mathit{put}(\{x{\in}a \mid p(x)\}, \{y{\in}c \mid p(y)\}) ::: r.\mathit{put}(\{x{\in}a \mid \neg p(x)\}, \{y{\in}c \mid \neg p(y)\})\\
&\quad \mathit{create}(a) \quad = l.\mathit{create}(\{x{\in}c \mid p(x)\}) ::: r.\mathit{create}(\{x{\in}c \mid \neg p(x)\})\\
&\}
\end{aligned}
$$

The *fork* lens combinator is one way of realizing *parallel lens composition*: in contrast to the *comp* lens combinator, which puts two lenses in sequence, *fork* puts two lenses in parallel. Using *fork*, other tree-specific lenses can be constructed. For instance, a *filter* lens can be created by applying $\mathit{const}(\{\})$, a lens which realizes a constant replacement with an empty tree, to the one subset and the *id* lens to the other subset. This way, the subset which does not satisfy the condition is filtered away, whereas the other subset stays untouched:

$$\mathit{filter}(p : \mathit{condition}) : \mathit{lens} = \mathit{fork}(p,\ \mathit{id},\ \mathit{const}(\{\}))$$

The introducing example in Foster et al. (2005) is a synchronization between the contact list tree shown above and a phone book tree which only contains names which refer to a phone number (Fig. 5.3). The lens which implements this synchronization therefore, in the forward direction, filters away the URLs and flattens the tree by one level. This lens can be described by parameterizing a *focus* lens to extract the phone number and composing it with a *map* lens combinator to apply it to all entries of a list:

$$phoneBookSync : lens = map(focus(\texttt{"Phone"}, \{\texttt{"URL"} \mapsto \texttt{"http://default.com"}\})$$

The *focus* lens again is composed: *focus*(n,d) can be expanded to *comp*(*filter*(n,{d}), *hoist*(n)) where $n$ is an edge-label and $d$ is an appropriate default structure for the *create* function – in the example, a default URL.

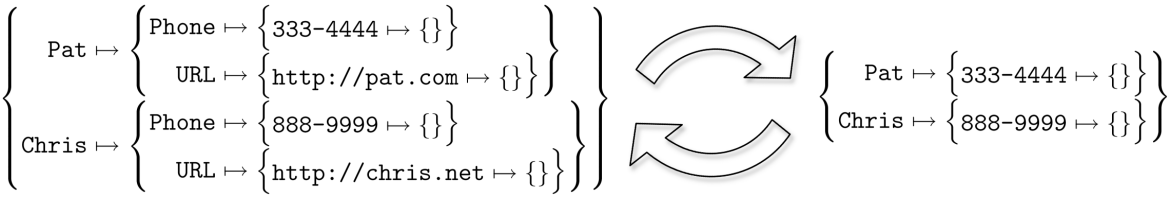



Figure 5.3: A concrete tree and a derived abstract tree being kept in sync by a lens

*Focal* demonstrates that with a comparably small set of atomic lenses and combinators a rich lens library can be constructed. Based on *Focal*, Bohannon et al. (2008) presented a comprehensive lens library for bidirectional string transformation called *Boomerang*.

### 5.1.2 Delta-Based Lenses

In the general lens concept which we presented so far, the lens functions take a potentially updated structure – a tree, a model, or a part of a model – as the input: In the *get* direction, the concrete structure is translated into a corresponding abstract structure; *get* is not incremental, i.e., if there has been a previous version of the abstract structure, it is simply discarded. In the *put* direction, the lens takes an updated version of the abstract structure and can use information from the original concrete structure for creating an updated version of the latter. In other words, *put* has to find out what has changed on the abstract side (update discovery) and then it has to figure out, using the original concrete structure, what the corresponding changes on the concrete side are. Because the lens does not know what has actually changed but only works with potentially updated structures – in other words, *states* – the lens approach presented so far has been termed *state-based* to differentiate it from the following approach.

With delta-based lenses, as introduced by Diskin et al. (2011b), a lens function takes the update itself (the delta) as the input and produces a corresponding update, which is applied to produce an updated structure. In contrast to state-based lenses, which translate a structure (i.e., its state) to another structure, delta-based lenses translate deltas to deltas. We already introduced this notion as the difference between discrete incrementality and trace-based incrementality in our taxonomic space in Sec. 3.2.4. Informationally asymmetric delta-based lenses, for instance, support synchronization type (101).

The backward transformation of a delta lens, *dput*, is defined as $dput : \Delta_A \rightarrow \Delta_C$. In contrast to state-based *put*, *dput* does not take an original concrete structure as input. Instead, the resulting $\Delta_C$ is applied to the original concrete structure $c$ to yield the updated concrete structure: $c'$: $\Delta_C(c) = c'$. Deltas can therefore be seen as homoge-

neous update functions so that lens functions of a delta lens can be seen as higher-order functions, i.e., functions processing functions. Then, *dput* for instance, can be defined as $dput : (A \to A) \to (C \to C)$. In contrast to state-based lenses, where the lens functions are actually transformations, it makes sense to call the lens functions of delta lenses more generally update propagation functions. Fig. 5.4 visualizes the structure of a delta lens.

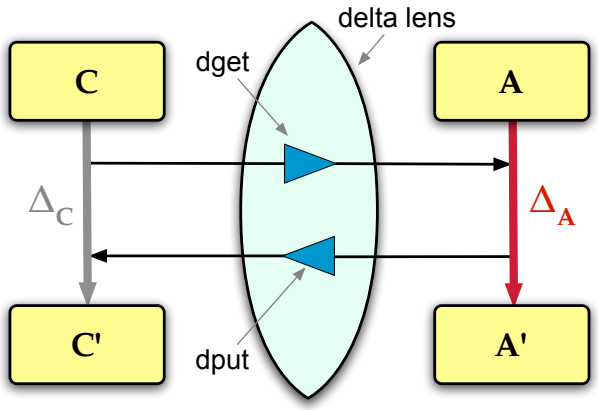



Figure 5.4: A delta lens propagates updates (deltas) instead of states

The updates which serve as input for a delta lens' functions have to be retrieved from somewhere. For instance, a modeling tool could record traces of edits when they are made, and then provide those traces to the lens. The updates can alternatively be retrieved by heuristics-based heterogeneous *model matching*, e.g., by comparing the previous state of the structure with the updated one, based on names or structural similarities. This way, the process of finding out what has been changed is decoupled from the process of translating those updates and can therefore be externalized. With state-based lenses in contrast, those two tasks are intermingled: each atomic lens or lens combinator contains both the logic to find out what has changed and where to integrate it.

In the context of delta lenses, it makes sense to think of updates as *vertical deltas* – elements are homogeneously mapped to updated elements on the same (abstract or concrete) side – and to think of the update translation as *horizontal deltas* – elements on one side are heterogeneously mapped to elements on the other side. The horizontal deltas can be obtained by heuristics-based heterogeneous *model matching* if both, the concrete and abstract structure already exist. Or, if the abstract structure does not exist yet, the horizontal deltas could be obtained initially by taking traces from a (state-based) forward transformation which creates an abstract structure from a concrete one. This initial creation process can also be interpreted in delta terms as providing an update from an *initial empty concrete structure* (denoted by $\Omega_C$) and translating this update to a creational update on the abstract side (i.e., from $\Omega_A$). This way, also the backward *create* function from state-based lenses can be realized. The delta-based forward and backward *create* functions are actually just special applications of the normal *dget* and *dput* functions, as long as they are able to handle the special empty structure. However, in the following definition of a delta lens' functions, we present the two create functions

explicitly. As can be seen, delta lenses are incremental in both directions by default and only non-incremental in the special *create* case, whereas with state-based lenses, only the backward transformation *put* implements discrete incrementality.

$$\begin{array}{lllll} \mathit{dget}: & (C \to C) & \to & (A \to A) \\ \mathit{dput}: & (A \to A) & \to & (C \to C) \\ \mathit{fwdcreate}: & (\Omega_C \to C) & \to & (\Omega_A \to A) \\ \mathit{bwdcreate}: & (\Omega_A \to A) & \to & (\Omega_C \to C) \end{array}$$

Similar to the general concept of state-based lenses, delta lenses are not tree-specific. An advantage of delta lenses concerning model transformations is that model matching can be externalized, i.e., tools such as *EMFCompare* could be used to obtain model deltas.

However, in the following section, we present the adaption and implementation of *Focal* as an internal Scala MTL. One reason for choosing a state-based transformation language is that seamless integration with existing modelware language workbench tools is one of our main goals, and tools such as *Xtext* always create a new model from scratch when it is edited; traces are not provided. Therefore, it requires more effort to integrate an MTL based on delta lenses with those tools. With state-based lenses, a model synchronization layer can be implemented which is completely decoupled from the editing tools as the synchronization works only on the models created by editing tools. The other reason is that delta lenses mainly exist as a conceptual framework – there are no delta lens libraries yet. In other words, there are no actual delta-based transformation languages comparable to *Focal* or *Boomerang*, which we could have adapted to our needs.

## 5.2 Object Tree – A Data Model for Modelware Lenses

For adapting *Focal*, we first have to look at its data model – edge-labeled trees – and how it can be adapted to a modelware setting. We therefore take the characteristic properties of the object-oriented modelware space into account but try to stay as close as possible to the semantics of *Focal* in order to be able to reuse many of its lenses and combinators. An object is a triple of a (1) unique identity by which it can be referenced, (2) a state, and (3) the implementing class defining valid operations on that object (Szyperski, 1999). The state of an object is defined by the values of a fixed number of fields. In a Java-based context, fields have a unique name and a static type. Fields containing multiple values can be expressed as a homogeneously typed collection, such as an indexed list or a key-value map.

Thus, if we want to model the lens synchronization example from 5.3 in an object-oriented fashion, we could come up with an object structure as presented in the UML object diagram in Fig. 5.5. By comparing this with the edge-labeled tree from the original example, we can infer in what way the data model needs to be adapted. In the following three subsections, we explain the main differences, which are, in short, distinction between meta- and instance-level-data, non-containment references, and order. Afterwards, we define our data model, which we call *object tree*, and present a type hierarchy for it.

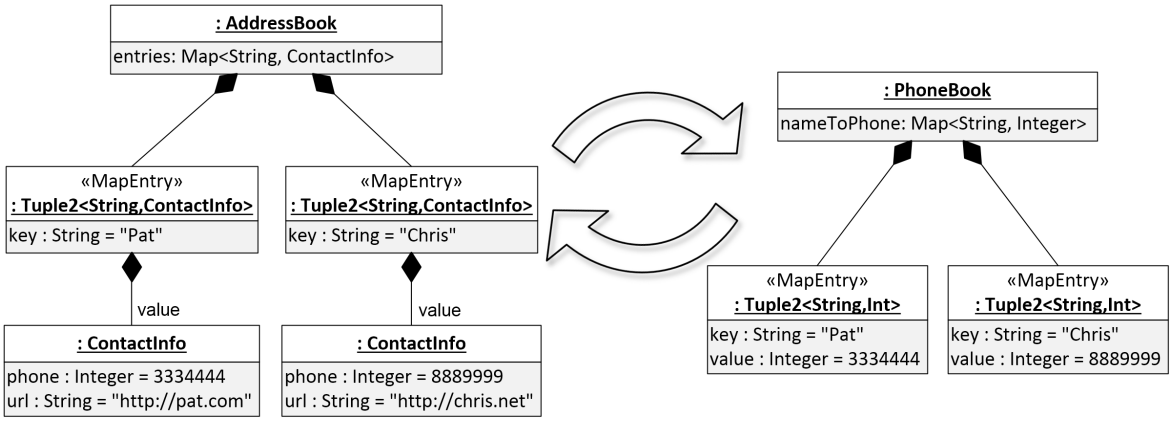



Figure 5.5: UML object diagram of an object-oriented version of the example from Fig. 5.3

### 5.2.1 Meta-Data vs. Instance-Data

First, in the edge-labeled tree, labels are used to access the children of a tree node. The counterparts in object-orientation are either class field names (or the names of public getter methods, respectively) or the index by which one can access a specific element in a homogeneously typed list (or the key to access a value in a key-value map, respectively). Whereas data in the edge-labeled tree is stored as labels – e.g., the phone number in the example – we cannot save data as a field name in Java or Scala. This reveals one of the key differences between the original data model and the required one: With the edge-labeled tree, we have no meta-layer but only an instance layer. Both meta-information, such as the description of the content of a field, and value information is mixed – both "phone" and "3334444" are labels. So, what is always a label in the edge-labeled tree, is in object-oriented terms sometimes meta-information and sometimes value-information. This means, we need both lenses that work on the meta-level and lenses that work on the instance level. For instance, there cannot be just one *fork* combinator as in *Focal*. We need one *fork* which splits the fields of an object into two sets based on the statically available meta-level field names, and one *fork* which splits the content of a collection based on value-information available only at runtime, e.g., based on the keys of a key-value map. In our data model, the distinction between meta- and instance-level data is reflected by every node having to refer to its meta-information by a type annotation. In contrast to the lenses in *Focal*, lenses designed for this data model are, in general, *two-level transformations* (Cunha et al., 2006): They transform the schema at the meta-level and, at runtime, transform instances at the value-level accordingly.

### 5.2.2 Non-Containment References and Leaf Representation

The next difference is that objects can reference other objects which are not considered their children, i.e., they can have non-containment references – object structures and models, in general, are graphs. However, as we explained in Sec. 2.3.1 Java-based modeling frameworks such as EMF, practically enforce a spanning containment tree within

a model's object graph. This constraint has been shown to be useful for graph traversal and persistency management. We include this constraint in our data model to allow the pragmatic adaption of *Focal* to the modelware setting. We consider a model as a containment tree with occasional non-containment references. In this tree of objects, values can actually be saved only as leafs, that is, as non-reference attribute values. In contrast, in the edge-labeled tree, data can also be stored as labels of nodes. In our data model, leafs are therefore not represented by normal tree nodes with an empty children lists but by special value-holding leaf items with no children list. This explains why in the example object structure above, value `"Pat"` has become a leaf whereas it has been a node in the edge-labeled tree shown in Fig. 5.3.

### 5.2.3 Ordered Children Lists

Finally, the children of a tree node in an edge-labeled tree are represented by a *set* of labels, i.e., they are unordered and without duplicates. Now, as we defined field names or collection indices/keys, respectively, as the counterparts to labels for accessing child elements, the uniqueness for children still holds true: indices or keys are unique by definition and field names in Java or Scala are also required to be unique in one class. Concerning order, the situation is more diverse: indices are obviously ordered but class fields and dictionary keys are generally considered unordered. However, EMF for instance, represents the children of a containment tree node as an ordered list for XML persistence reasons. Furthermore, the fields of Scala case classes coincide with the argument list of the class' constructor, which is also ordered. Thus, for uniformity, we define the children of a node to be an ordered list without duplicates and define the order of fields to be alphabetical. This means that in our data model an edge-label becomes an index of this contained-children list. If the containing node is not a collection, an index can be mapped to a field name using information from the node's type annotation. Note that the uniqueness constraint only applies to the labels, i.e., to the list indices which are by definition unique, and not to what they refer to, i.e., the actual children.

### 5.2.4 A Type Hierarchy For Object Trees

Summarizing, we call this data model, which we designed as a pragmatic adaptation of an edge-labeled tree to an object-oriented modelware context, an *object tree*.

*Definition* 5.1 (object tree). An *object tree* $\mathcal{T} = \langle t, id, [v|l] \rangle$ has a type-annotation $t$, a unique identity $id$, and contains either a single value $v$ or an ordered list $l$ referring to either a fixed number of subtrees (the fields) or an arbitrary number of subtrees of the same type (the contents of a collection). Single value tree nodes can represent a non-containment reference by holding the id of another tree node within the same tree.

This data model is partly inspired by the *ATerms* format presented by van den Brand et al. (2000). We therefore call the basic type of our object tree data model a *term*. In its most general form, a term is a tree node or a tree leaf with meta-information attached. As we have seen in the example above, we need different types of terms to represent

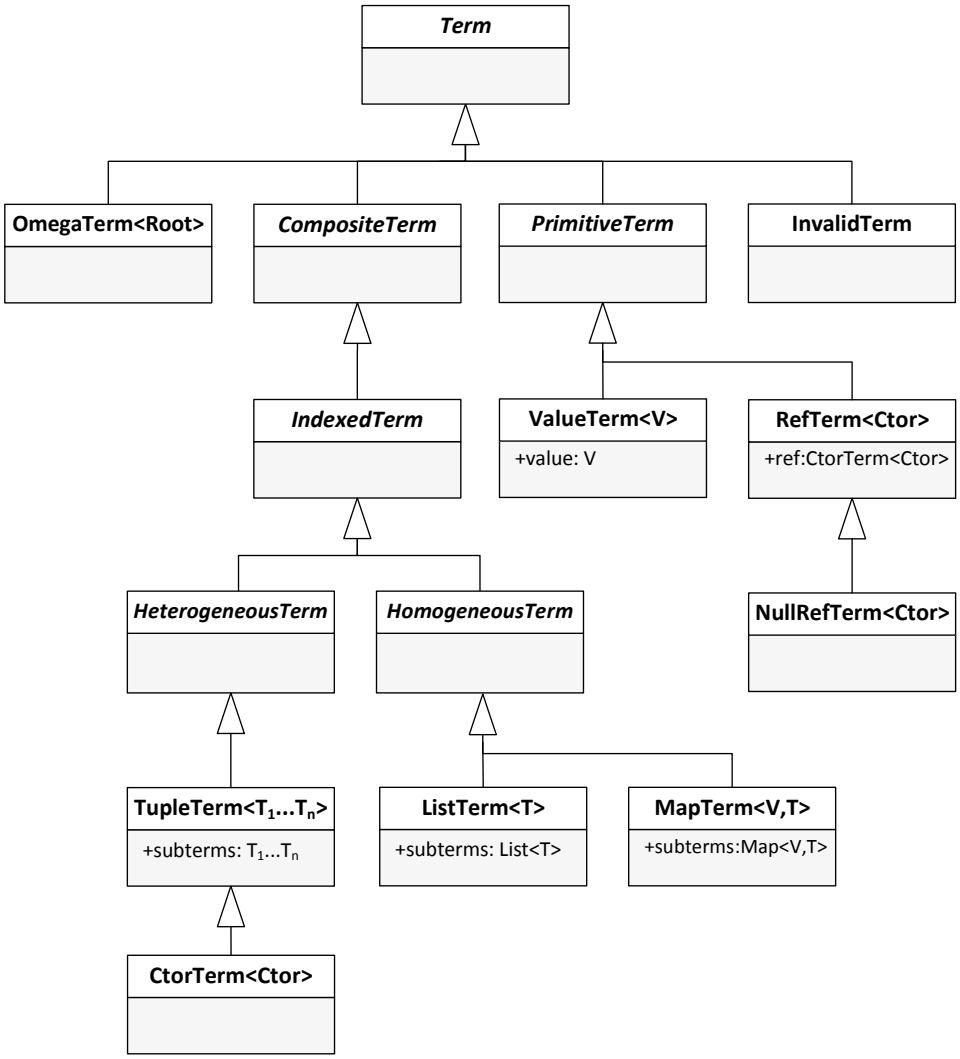
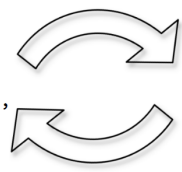



Figure 5.6: A meta-type hierarchy for the object tree data model

a typical object-oriented data structure. The most important is the *constructor term* (*CtorTerm*) which represents reference type instances. A constructor term has a fixed number of child terms with potentially different types, and refers to its class type *Ctor* – its arity and the order of its subterms is determined by its class type. Next, there are two *homogeneous collection terms*, i.e., where all subterms have the same type: a *list term* (*ListTerm*) and a key-value *map term* (*MapTerm*). The size of a homogeneous collection term is not statically fixed but can change at runtime. In contrast, a *tuple term* (*TupleTerm*) is heterogeneously typed term with a statically fixed number of subterms. In contrast to a constructor term, of which it is a generalization, a tuple term does not correspond to a class type. It is used, for instance, to represent the key-value pairs which are the children of a map term. Then we have a *value term* (*ValueTerm*) to represent

the leaves in the tree which carry the actual data. As a special kind of value term, there is a *reference term* (*RefTerm*) which contains a non-containment reference to another constructor term. Furthermore, there is a special term which represents an empty model (*OmegaTerm*), and therefore refers to the model's containment hierarchy's root type. Fig. 5.6 presents the hierarchy of meta-types which we use to represent models with our object tree data model. Type parameter bound are omitted for brevity – $T$ always refers to another term type. Fig. 5.7 shows a representation of the models from Fig. 5.5 using our term types – type annotations are enclosed in square brackets; the children of a term are listed in normal round parentheses, separated by commas.

```
CtorTerm[AddressBook](
 MapTerm[String,CtorTerm[ContactInfo]](
  TupleTerm(
   ValueTerm[String]("Pat"),
   CtorTerm[ContactInfo](
    ValueTerm[Integer](3334444),
    ValueTerm[String]("http://pat.com"))),
  TupleTerm(
   ValueTerm[String]("Chris"),
   CtorTerm[ContactInfo](
    ValueTerm[Integer](8889999),
    ValueTerm[String]("http://chris.net")))))
```



```
CtorTerm[PhoneBook](
 MapTerm[String,Integer](
  TupleTerm(
   ValueTerm[String]("Pat"),
   ValueTerm[Integer](3334444)),
  TupleTerm(
   ValueTerm[String]("Chris"),
   ValueTerm[Integer](8889999))))
```

Figure 5.7: The models from 5.5 represented in a type annotated term notation

Comparing the object tree data model with that of edge-labeled trees, type-annotations and implicit object-ids were added, and order of subterms now matters. Edge-labels are replaced by indices, which – in the case of a constructor term – can be mapped to field names. The term type hierarchy will help us to express type constraints on the data that a lens can handle. Together, the data model and the type hierarchy allow us to implement most of the lenses of *Focal* with similar semantics, and at the same time allow us to define further lenses which are required in an object-oriented modelware setting.

## 5.3 Type-Safe Object-Tree Lenses in Scala

In this section, we embed *Focal* as a type-safe internal DSL into Scala. We implement lenses from *Focal* in such a way that they can handle data which conforms to our object tree data model. This way, we ensure their general applicability in a modelware setting. However, in this section, we do not provide any special lenses for model transformation. We will do this in the next section. Instead, we concentrate on achieving a similar semantics and syntax as *Focal*, and on achieving static verification of lens composition using Scala's type checking. The latter can be of great help when composing complex lenses and therefore fits well with the combinator-based approach of lenses in general.

This section is structured as follows: In the next subsection, we present our general approach to the embedding of *Focal* into Scala, and how lens composition can be type-checked. In Sec. 5.3.2, we show how any Java- or Scala-based domain object can be implicitly converted to an object tree while keeping all type information. In Sec. 5.3.3, we imple-

ment lenses which are explicitly type-parameterized in order to allow their composition to be statically type-checked. In Sec. 5.3.4, we present an alternative, more advanced approach of fully generic lenses which do not need to be explicitly type-parameterized, and still allow type-checked composition. However, because of other disadvantages, we decide not to use this approach in the rest of this dissertation, but instead alleviate shortcomings of the explicit-type-parameterization approach by introducing type-inferring lens composition operators. This is presented in Sec. 5.3.5. Afterwards, we conclude the section.

### 5.3.1 Towards a Type-Safe Lens Language

We want to create a concise internal DSL syntax to parameterize, compose, and apply lenses. At the same time, we want to provide as much tool assistance as possible. The DSL therefore has to preserve static type information, so that the Scala compiler can type-check lens composition and lens application. However, lots of type-annotations can make an internal DSL less concise and less comfortable to use. We try to achieve both conciseness and type-checking, with the help of Scala's type inference.

The following listing shows an abstract generic type for asymmetric state-based lenses which process terms according to our object tree data model, and translate between a concrete term of type $C$ and an abstract term of type $A$. Both types $C$ and $A$ have

Listing 5.1: A generic type for asymmetric state-based lenses

```
abstract class Lens[C <: Term, A <: Term] {
  def get(c: C): A
  def put(a: A, c: C): C
  def create(a: A): C
}
```

to inherit (expressed by `supertype <: subtype`) from type `Term`, the root type of our term type hierarchy. Based on this general lens interface, we can define simple lenses and lens combinators such as the sequential composition *comp* (see Sec. 5.1.1). The following listing shows how to express *comp*'s type constraint which says that the abstract term of the left lens $l$ must be of the same type (here `AC`) as the concrete term of the right lens $k$.

Listing 5.2: A combinator for type-safe sequential lens composition

```
class Comp[C,AC,A](l: Lens[C,AC], k: Lens[AC,A]) extends Lens[C,A] {
  def get(c: C): A = k.get(l.get(c))
  def put(a: A, c: C): C = l.put(k.put(a, l.get(c)), c)
  def create(a: A): C = l.create(k.create(a))
}
```

Because we expressed the type constraint in terms of type parameters, the Scala compiler can check whether two lenses are compatible for sequential composition when creating a new *comp* instance. If the lens types are not compatible, a compile-time error will occur and any standard Scala editor will highlight the composition as errorneous.

To show how the lens DSL is intended to be used for an actual synchronization task, let the domain classes and the concrete example tree structure from the phone book example (see Fig. 5.5) be implemented as in the following listing (using Scala's case classes).

Listing 5.3: Definition of the domain classes and the example tree from Fig. 5.5

```
1 case class AddressBook(entries: Map[String, ContactInfo])
2 case class ContactInfo(phone: Int, url: String)
3 case class PhoneBook(entries: Map[String, Int])
4 val ab = AddressBook(Map("Pat" -> ContactInfo(3334444,"http://pat.com"),
5                          "Chris" -> ContactInfo(8889999,"http://chris.net"))
```

Now, the goal of the DSL is to enable parameterization and composition of pre-defined lenses from a lens library (here, *focus* and *map*), and to use the resulting composed lens to transform domain objects as in the following listing, while keeping static type-safety.

Listing 5.4: Envisioned usage of the internal lens DSL, similar to *Focal*'s usage (see p. 107)

```
1 val ab2pb = Map(Focus(phone)) // composing and parameterizing the lens
2 val pb: PhoneBook = ab2pb.get(ab) // derive abstract tree as phonebook
3 pb.entries("Pat") = 3334321 // modify the contents of the phonebook
4 val abnew: AddressBook = ab2pb.put(pb, ab) // put the changes back
```

Note, that when composing a lens as shown in line 1, we do not want to specify the specific domain class types as type arguments because this would clutter the lens composition. On the other hand, we want to check at compile-time whether the address book instance `ab` (passed in line 2) is a valid input for the *get* function of the composed `ab2pb` lens (read: address book to phone book lens), and that the result will be of type `PhoneBook`. Now, the problem is that we want to directly apply lenses to domain objects as shown above (lines 2 & 4), but the lens functions expect a subtype of `Term` which the domain classes do not extend. However, we do not want our DSL to require users of the DSL to modify domain classes. Therefore, we need to convert domain objects to term objects, while preserving type information.

### 5.3.2 Converting Domain Objects to Typed Terms

In order to be able to implement pre-defined lenses – atomic lenses and lens combinators – independently from domain classes, these lenses need to be defined on general term types. However, to apply these lenses directly on domain objects, domain objects have to be converted to terms. We use Scala's implicit conversions to transparently convert domain objects to terms and vice versa. We want to preserve static type-safety throughout the whole transformation process. We therefore have to keep track of the types of all of a term's subterms. This typing cannot be achieved only by annotating constructor terms with a corresponding class type because within the transformation process *intermediate term structures* can emerge that do not correspond to any source or target domain class – for instance, when splitting up a source domain object, the results of this splitting needs to get a type before putting things together to yield a target domain object.

Because Scala's type system – like most common type systems – provides either a heterogeneously typed tuple construct with a fixed arity (e.g., `Tuple3[A,B,C]`) or a homogeneously typed collection (e.g., `List[A]`), we use *heterogeneously typed lists* (HLists) as introduced by Kiselyov et al. (2004) for intermediate term structures. HLists are based on type-parameterized *Cons-cells* and can be implemented and used in Scala as follows.

Listing 5.5: A simple Scala implementation of a heterogeneously typed list

```
1 abstract class HList // base type with the following two subtypes:
2 case class HCons[H, T <: HList](head: H, tail: T) extends HList // the cons cell
3 case class HNil extends HList // type to express the end of a list
4 val hl: HCons[String, HCons[Int, HNil]] = HCons("str", HCons(42, HNil)) // usage
```

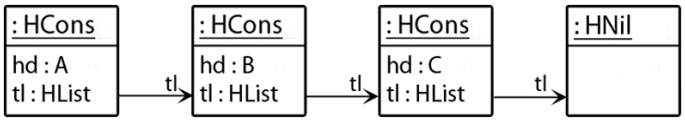



Figure 5.8: Object diagram of an HList which contains values of type A, B, and C

As can be seen in line 4 of the above listing, heterogeneously typed list instances can now be defined with static type-safety – a compile-time error will occur if the inferred type of the HList instance at the right does not match the type annotation at the left. However, both the nested type annotation and the nested list instantiation is not very concise. Therefore, some Scala implementations of HList – e.g., J. Nordenberg's implementation[1] – define *type list* types (TList) correspondingly and define the type alias `::[H,T]` for `TCons[H,T]`. Then, using a TList as the type parameter of HList allows the concise definition of a list instance that contains objects of type A, B and C as `HList[A::B::C::TNil](a,b,c)`. A constructor term wraps such an HList and has two type parameters: the corresponding class type *Ctor*, and *TL*, the type list of its inner HList. This way, a constructor term can contain subterms of different types and still keep their type information. Domain objects can be converted back and forth implicitly. Therefore, pairs of conversions have to be provided for every domain class whose objects may be involved in a synchronization. The following listing shows the definition of such an HList-wrapping constructor term class, and the signatures of the two implicit conversion functions which are required to implicitly convert a `ContactInfo` domain object to a correspondingly type-parameterized `CtorTerm` object and vice versa.

Listing 5.6: Type information preserving conversion between domain and term objects

```
1 class CtorTerm[Ctor, TL <: TList](subterms: HList[TL])
2 // implicit conversions for ContactInfo:
3 implicit def ci2term(ci: // from ContactInfo to CtorTerm
4   ContactInfo): CtorTerm[ContactInfo, ValueTerm[Int] :: ValueTerm[String] :: TNil] = ..
5 implicit def term2ci(t: // from CtorTerm to ContactInfo
6   CtorTerm[ContactInfo, ValueTerm[Int] :: ValueTerm[String] :: TNil]): ContactInfo = ..
```

Fig. 5.9 visualizes how, using the above implicit conversions, a `ContactInfo` object from the example in Fig. 5.5 is converted into a corresponding `CtorTerm` object and vice versa (for simplicity, on the term side, values are not wrapped in value terms).

For every domain class two such implicit conversion definitions need to be provided for the lens DSL to work as demonstrated. However, those conversion definitions can be generated automatically. We already presented the general approach of automatically generating source code of case class definitions and implicit conversions in the last chapter for our unidirectional MTL (see Sec. 4.3.3). We use a similar approach here, but this

[1] http://jnordenberg.blogspot.com/2009/09/type-lists-and-heterogeneously-typed.html

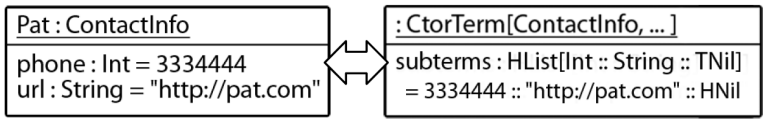



Figure 5.9: Conversion between domain objects and type-parameterized terms

time we only generate the conversion definitions. We cannot generate specific term type definitions such as `ContactInfoTerm` which correspond to domain classes (as we did with case classes in Chap. 4) because we would also need to generate the definitions of all possible intermediate term types. As intermediate terms could be any combination of domain class corresponding terms, this would result in an explosion of type definitions. Instead, we only generate two implicit conversions for every domain class and handle intermediate terms by type-parameterizing the generic tuple term type, which does not need to refer to a domain class type. Because this type-parameterization can be done implicitly using type inference, intermediate terms preserve all type information, but no explicit definition or parameterization is required.

### 5.3.3 Type-Parameterized Lenses

Now that we have term types which allow the type information of their domain class counterparts to be preserved, we can start to implement a reusable library of pre-defined atomic lenses and lens combinators which are defined on those term types. However, in spite of the lens library being defined independently from actual domain classes, we want to type-check whether a given lens – which may be composed out of many small sublenses – conforms to the structures it is intended to synchronize.

#### A Simple Type-Safe Lens

With a lens which is not parameterized – such as the *hoist* lens, where the structural modification is always the same – the two types $C$ and $A$ between which a lens translates, only depend on each other. *Hoist*'s $C$ is always a type of a term with one single edge at the root (this is the $C$-side constraint of the *hoist* lens) and $A$ is always the type of the single child that this edge refers to. Thus, the type list of term type $C$ is a list of length 1 with type $A$ as the only component at position 0. This is written as `A::TNil`, which denotes a type $A$ being prepended to the end of a type list, `TNil`. Type $C$ can be described as `TupleTerm[A::TNil]` – a constructor term is also a tuple term, so defining hoist on tuple terms is more flexible. Thus, the type of the *hoist* lens is `Lens[TupleTerm[A::TNil], A]` extending the generic lens type `Lens[C <: Term, A <: Term]`. So the only free type-variable of *hoist* is $A$. The following listing shows the complete Scala definition of a type-safe *hoist* lens which processes tuple terms.

$A$ is the only type parameter of *hoist* (line 2) and the *hoist* class extends the basic lens type from Listing 5.1. In line 3, `C` is introduced as a *type alias* for `TupleTerm[A::TNil]` which makes the definition of the three lens functions shorter. Now, when type parameter $A$ of *hoist* is set to a specific term type (such as `TupleTerm[ValueTerm[String]::TNil]`)

Listing 5.7: A type-parameterized type-safe *hoist* lens class

```
1 // an atomic hoist lens which removes the single edge at the root of a tree
2 class Hoist[A <: Term]() extends Lens[TupleTerm[A::TNil], A] {
3   type C = TupleTerm[A::TNil] // constraints possible concrete terms to this shape
4   def get(c: C): A = c.subterms.head // returning the head of the list of subterms
5   def put(a: A, c: C): C = this.create(a) // oblivious lens: put = create; c not needed
6   def create(a: A) = TupleTerm(a::HNil) // adds the single edge '_0 -> a'
7 }
```

$C$ is determined by $A$. Thus, *hoist*'s lens functions are statically typed and their usage can be type-checked.

### Type-Level Parameters and Abstract Type Members

When a lens is further parameterizable – such as the *focus* lens which accepts an edge label to focus on – we can type-check a parameterized lens instance only if the parameter is also specified at type level. To achieve this, some basic *type-level programming* is required. We already showed in the previous code listing how a type alias can be introduced using the keyword `type` inside a class declaration. Such a type declaration inside a class is called a *type member* and is considered a member of that class, like methods and fields. A static type member – that is, a type member which is the same for all instances of a (possibly type-parameterized) class – is accessed using `#`, which is called a *type projection*. For instance, in the previous example, type member `C` of the `Hoist` class can be accessed by `Hoist#C`. Like other members, type members can be declared as *abstract type members* in an abstract class (or an interface) and can be implemented by subclasses. As type members can have type parameters themselves, they can be used to realize *type functions* which are evaluated at compile-time. The declaration `type Fun[X <: Dom] <: CoDom` defines an abstract type function that has one parameter, which has to be a subtype of a type `Dom`, and that evaluates to a subtype of a type `CoDom`.

Because the tree lenses which we implement primarily use edge labels as parameters, and because edge labels in our object tree data model are translated to indices (which represent field names or collection indices), we need to encode indices – in other words, natural numbers – as Scala types. Such type-level numbers can be implemented as recursively nested successors of a bottom type, which in this case obviously is the number 0. Numbers encoded this way are called *Peano numbers*[2]. In Scala, type-level Peano numbers can be implemented by declaring an abstract type `Nat` for natural numbers, and a successor class `Succ[P <: Nat](i: Int)` which extends `Nat`. Then a type-level number literal can be declared as `type _1 = Succ[_0]` with a corresponding instance-level number literal being defined as `object _1 = new Succ[_0](1)`. If such a number literal is passed to a generic instance-level function, the number type can be inferred and is therefore available at compile-time (refer to type parameter inference, Sec. 2.4.6).

With these number types and number literals we can define type-safe methods of `HList`, such as a type-safe indexed accessor method called `nth` as presented in Listing 5.8. In line 2, `Nth[N]` is declared as a type function of `TList`. In line 6, this type function is used

[2]http://www.haskell.org/haskellwiki/Peano_numbers

Listing 5.8: Defining a type-safe indexed accessor method for HLists and tuple terms

```
1  abstract class TList {
2    type Nth[N <: Nat] // TList's abstract type member / type function yielding nth's type
3    ...
4  }
5  abstract class HList[TL <: TList] {
6    def nth[N <: Nat](n: N): TL#Nth[N] // type-safe indexed accessor on HList instances
7    ...
8  }
9  class TupleTerm[TL <: TList] {
10   val subterms: HList[TL]
11   type Nth[N <: Nat] = TL#Nth[N]
12   def nth[N <: Nat](n: Nat): Nth[N] = subterms.nth(n)
13 }
```

by `HList`'s `nth`-method to determine the result type of accessing the $n$th element of the list. This `nth`-method expects an instance-level number literal $n$ of type $N$ and is used like this: `val x = myhlist.nth(_2)`. Type $N$ is inferred from the type of the passed instance-level number literal, so that the type parameter does not need to be specified explicitly ,although one could specify it explicitly by writing `myhlist.nth[_2](_2)`. This has the advantage, that the instance-level literal $n$ can be passed to `nth` from somewhere else, which is what our tuple term implementation does. It exposes a type member `Nth` which is forwarded to its type list, and an `nth` method which is forwarded to its inner heterogeneous list of subterms (lines 11 & 12).

### Lens Parameterization and Composition

With this framework of implicit conversions, term types, number types, and type-safe operations on heterogeneous lists, we can define some more interesting, parameterized lenses. As an example, we define an atomic *filter* lens which is parameterized with a single index – in *Focal*, *filter* takes a set of labels instead, and is implemented as a parameterized fork lens. To distinguish them, we call our variation `FilterN` as it takes a single index $n$. However, the semantics is similar to the original *filter* lens: In the *get* direction, all direct children except the specified one are filtered away, so `FilterN.get` always returns a tuple term with a single child. There is a subtle difference in semantics as the single remaining child is the only component of the abstract tree's child list and therefore always gets index 0. Technically, the label is changed from $n$ to 0 in the *get* direction because our data model translates label sets to indexed lists. As the lens knows about the specified index, it can reintegrate changes correctly in the *put* direction, so that this subtle semantics change has no practical consequences. The following listing shows the definition of `FilterN` and shows how the *focus* lens which we envisioned in Listing 5.4 can be defined by sequentially composing `FilterN` with the `Hoist` lens which we defined in Listing 5.7.

`FilterN` has two type parameters: the number type $N$ for the specified index and type $C$ of the concrete object tree. Type $A$ does not need to be specified because in this lens, $A$ is completely determined by $C$. $A$ is a tuple term with $C$'s $n$th subterm type as the type of the only child. This type is expressed using the type function `Nth` of tuple term

Listing 5.9: Composing the *focus* lens from the *filter* lens and the *hoist* lens

```
1 // an atomic filter lens which discards all but one (index-specified) child:
2 class FilterN[N <: Nat, C <: Term](n:N, d:C) extends Lens[C, TupleTerm[C#Nth[N]::TNil]]{
3   type A = TupleTerm[C#Nth[N]::TNil]
4   def get(c: C): A = TupleTerm(c.nth(n) :: HNil) // using the typesafe nth()-accessor
5   def put(a: A, c: C): C = c.replace(n, a.nth(_0))
6   def create(a: A) = d.replace(n, a.nth(_0)) // using the default term d instead of c
7 }
8 def Focus[N <: Nat, C <: Term](n: N, d: C) = Comp( FilterN(n, d), Hoist[C#Nth[N]]() )
9 // inferred type of Focus is Lens[C, C#Nth[N]], i.e. syncing between C and its nth child
```

which we defined in Listing 5.8. Thus, in Listing 5.7, the type of `FilterN` is `Lens[C, TupleTerm[C#Nth[N]::TNil]]` (line 2). In the *get* function (line 4), the type-safe accessor method `nth` is used – the result type of `c.nth(n)` is `C#Nth[N]`. In the *put* function (line 5) another type-safe method `replace` is used for reintegration. If there is no original concrete structure, the *create* function (line 6) uses a default $C$-side term $d$ instead of $c$. Therefore, `FilterN` expects $d$ as a second instance-level parameter, in addition to the index-parameter $n$.

Now, when composing the *focus* lens (line 8), the structure of type $A$ of `FilterN` (line 3) matches the structure of type $C$ of `Hoist` (see Listing 5.7). This satisfies the type-encoded constraint of the `Comp` lens and allows us to type-safely compose those two lenses. The composed `Focus` has the same type-level and instance-level parameters as `FilterN` to which they are passed – the type parameters are implicitly inferred. However, the type parameter $A$ of `Hoist` still has to be specified explicitly; we will show later how lenses can be composed without any explicit type parameterization. Figure 5.10 visualizes how the composed `Focus` lens works in the *get* direction when, for example, parameterized with index 1 and applied to an exemplary tree with index-labeled edges.

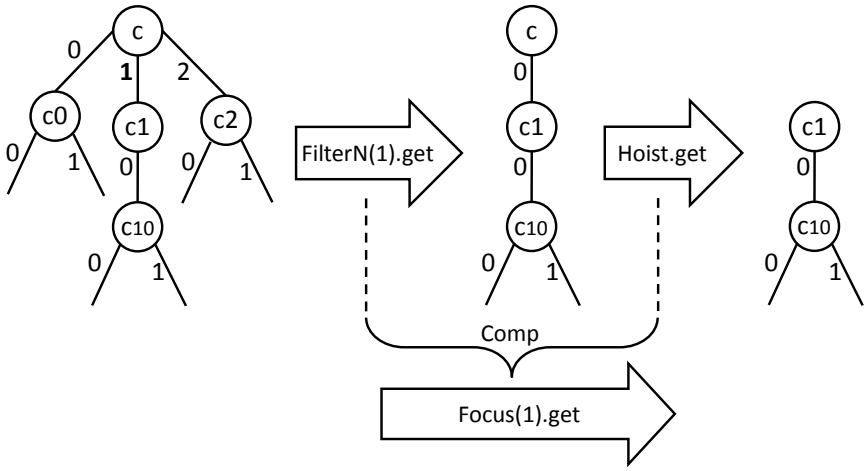



Figure 5.10: *Focus* as a composition of *filter* and *hoist*

Now, we want to use the composed *focus* lens to extract the phone number of a `ContactInfo` object, as we have envisioned it in Listing 5.4. However, for being able to directly use the lens with domain objects (and not term objects), we need to change

the definition of `Focus` slightly. For the conversion from a domain object to a corresponding term object to be triggered implicitly, the lens functions have to expect objects of a constructor term type and not of the general abstract term type. This is because our generated implicit conversions always convert from or to constructor terms. In the following listing, we first show a slightly altered definition of `Focus` which expects a constructor term as the default term, and infers the type of the type list of term type $C$ instead of inferring type $C$ itself. Afterwards, we parameterize `Focus` with the `_0` number literal and use it to extract the phone number – the 0th member – of a `ContactInfo` object.

Listing 5.10: Using the *focus* lens to extract the phone number of a contact info object

```
1  //defining focus so that it specifically expects CtorTerms
2  def Focus[N <: Nat, TL <: TList](n: N, d: CtorTerm[_,TL]) = Comp(...)
3
4  // a Focus instance to extract the phone number at index 0:
5  val focusCI_0 = Focus(_0, d=ContactInfo(42,"http://"))
6  // the type of this (instance-level) parameterized lens instance is inferred to:
7  // Lens[ CtorTerm[ContactInfo,ValueTerm[String]::ValueTerm[Int]::TNil], ValueTerm[Int] ]
8
9  // using the pre-typed lens instance
10 val cinfo = ContactInfo(3334444, "http://pat.com") // here, the concrete domain object
11 val phone: Int = focusCI_0.get(cinfo) // retrieving the abstract structure, type-checked
12 val ciNew: ContactInfo = focusCI_0.put(3334321,cinfo) // put changes back
```

As can be seen when we instantiate a parameterized `Focus` instance in line 5, the type parameters of `Focus` do not need to be specified explicitly. They are automatically inferred by the compiler: type $N$ is inferred from the type of the instance-level number literal $n$ (here `_0`) and type $C$ (more specifically the type list type $TL$ of $C$) is inferred from the type of the passed default object $d$ of type `ContactInfo` by looking up the corresponding domain-object-to-term implicit conversion. In line 7, the automatically inferred type of the parameterized `Focus` instance `focusCI_0` is shown (read: focus on `ContactInfo` field number 0). Because the inferred type $C$ of this lens instance is a parameterized constructor term type which corresponds to the `ContactInfo` domain class, we can now use the lens directly on domain objects: In lines 10–13, we first create a `ContactInfo` object `cinfo`, then use the *get* function to extract the phone number from it, and finally reintegrate a changed phone number back into a `ContactInfo` object (which is a new one because the lens functions are side-effect free). The `ContactInfo` object is automatically converted into a corresponding constructor term object and vice versa. Note that the conversion between value terms and their values is also done implicitly and type-safe, here between `ValueTerm[Int]` and `Int`. Internally, `Focus` first filters away the `URL` member of `ContactInfo` (more specifically the 1st component of the corresponding constructor term's child list) using the `FilterN` lens and then uses `Hoist` to extract the only remaining child, the phone number value term, which is then converted to an `Int`. All of this is completely type-checked and any Scala IDE would report errors if any of the (optional) type annotations do not match with the lens function's input/output types.

To conclude: because of implicit conversions to and from domain objects, the lens type (not the parameterized instance) could be specified independently from domain types but the type-parameterized lens instance can be used directly on domain objects. This

was one goal for the design of a tree lens library in Scala. However, in contrast to the desired syntax as shown in Listing 5.4, the *focus* lens is parameterized with an index instead of a field name.

### 5.3.4 Fully Generic Lenses

For some lenses – such as the *focus* lens – the approach of type-parameterization presented so far works well. However, for each class whose objects are to be transformed, a parameterized lens has to be instantiated. For instance, in the previous section we parameterized *focus* to extract the first child of a `ContactInfo` object. This type-parameterized lens instance can only be used with `ContactInfo` objects – it cannot be used to extract the first child of objects of other classes because the abstract type $A$ (in the phone example: `Int`) is determined at the time the lens is parameterized and instantiated. We call this kind of lens instance *pre-typed* because its types are fixed after instantiation, and the lens has to be typed before using it.

This pre-typing approach becomes an issue if, for instance, one lens is to be applied to one subterm and another lens is to be applied to all other subterms. In this case, differently typed instances of the latter lens have to be provided for each type of subterm. This applies already to very simple examples. In *Focal*, the basic *filter* lens is implemented by parameterizing the *fork* lens: The deletion lens *const({})* is applied to one subset of a tree's subtrees to filter them out, and the non-modifying *id* lens is applied to the other subtrees to keep them. However, with a heterogeneously typed term such as a constructor or tuple term, applying the pre-typed approach would require to provide a parameterized instance of *const* or *id* for each type which occurs in the list of direct subterms. Using a common supertype to pre-type a lens (e.g., `Id[Term]`) does not help either, because this way the specific type information of the different subterms is not preserved and therefore useful type-checking is not possible.

#### Inferring the Concrete Type and Computing the Abstract Type

To solve this issue, we developed lenses which are even more generic than pre-typed type-parameterized lenses. We call them *fully generic lenses* and describe them with a new abstract lens type called `GenericLens`. The main difference to the lens type presented in Listing 5.1 is that now the lens functions have a type parameter themselves which is automatically inferred from the passed function arguments. This way, type $C$ of the concrete term is not determined until – at compile-time – a lens' function is called. The type $A$ of the abstract term is then determined by a type function to express $A$ in relation to $C$. This type function `A[C <: Term] <: Term`, which computes one term type from another, is an abstract type member of the abstract `GenericLens` type, and must be implemented by concrete lens types such as `FilterN` etc. The following listing shows a snippet of the `GenericLens` interface with the type function for $A$, and the modified signature of lens function *get* which uses this type function.

```
1  type A[C <: Term] <: Term // type function to derive A from C
2  def get[C](c: C): A[C] // C is inferred and determines A
```

Now, when *get* is used, type $C$ is inferred from the passed instance-level argument $c$ (line 2) and it is type-checked whether the result type of *get* matches with `A[C]`. With this fully generic approach, a concrete lens type does not specify the relation between its types $C$ and $A$ by type parameters which have common (sub-)type parameters like, for instance, in the pre-typed `FilterN` (also shown in Listing 5.9):

```
FilterN ... extends Lens[C, TupleTerm[C#Nth[N]::TNil]] { ... }
```

Instead, the relation between $C$ and $A$ is described as a type function which maps from one term type to another; in the case of `FilterN` by:

```
FilterN ... { type A[C<:Term] = TupleTerm[C#Nth[N]::TNil] ... }
```

### Encoding Concrete-Side Constraints at Type-Level

An issue with describing type $A$ as a type function from $C$ to $A$ is that it is difficult to define constraints regarding the shape of type $C$. We already saw an example of such a $C$-side constraint in the *hoist* lens, where the concrete term must have exactly one child. In the pre-typed version of *hoist*, we encoded this constraint by `...extends Lens[TupleTerm[A::TNil], A]`. With a fully generic lens, we first infer type $C$ (instead of specifying it explicitly) and compute $A$ from it. We could define two sorts of generic lenses: those that express $A$ in relation to $C$ and those that express $C$ in relation to $A$. However, to be able to type-check composition of generic lenses, we need one common supertype and therefore have to decide for one direction of type inference and computation. Because our lenses are informationally asymmetric with a discrete incremental *put* function, type $C$ occurs in the parameter list of both *get* and *put*. Therefore it makes more sense to use type inference and computation from $C$ to $A$.

We then implement $C$-side constraints by a *boolean type function*. For this, we need (1) a supertype `Bool` for type-level booleans, (2) two boolean type-level literals `True` and `False`, and (3) type functions which evaluate to boolean types such as, for instance, a comparison type function defined on type-level numbers: `Nat#Equals[N<:Nat]<:Bool`. Furthermore, we need the ability to test for type equality to ensure boolean $C$-side constraints at compile-time. For this, Scala provides a type operator `=:=` which is a type alias for a type `Equal[A,B]`. A value-level function which has an implicit parameter of such a parameterized equality type, is automatically provided with an argument of that type if equality of the two compared types can be proved by the Scala compiler. If however type equality cannot be proved, type-checking fails because of a missing implicit argument, and the Scala compiler will report that it was not possible to prove the specified type equality. We add such type equality expression as an implicit parameter to the lens functions, and define $C$-side constraints as a type member of a lens. Using this type member in the type equality expression, we can then check $C$-side constraint on the basis of the type $C$ which is inferred when (at compile-time) a concrete argument is passed to a lens function.

The following listing shows the abstract generic lens type `GenericLens`. We define type bounds (individual for type $C$ and type $A$) as type parameters `CBound` and `ABound` of

Listing 5.11: An abstract type for fully generic lenses

```
1 abstract class GenericLens[CBound <: Term, ABound <: Term] {
2   type Constraint[C <: CBound] <: Bool // type function for defining constraints on C
3   type A[C <: CBound] <: ABound // type function to derive A from C
4   def get[C <: CBound](c: C)(implicit check: Constraint[C]=:=True): A[C]
5   def put[C <: CBound](a: A[C], c: C)(implicit check: Constraint[C]=:=True): C
6   def create[C <: CBound](a: A[C])(implicit check: Constraint[C]=:=True): C
7 }
```

class `GenericLens` (line 1). These type bounds are then used by the boolean type function for $C$-side constraints (`Constraint[C]` in line 2), and in the type function for computing type $A$ from type $C$ (`A[C]` in line 3) for restricting the input and output types of these type functions. The three lens functions then use type function `A[C]` to determine type $A$ from the inferred function type parameter $C$, which is also restricted by the $C$-side type bound. Note that the backward function *create* (lacking a concrete term parameter $c$) infers type $C$ by *create*'s result type, which is also supported in Scala. Furthermore, all three lens functions have an additional implicit parameter `check` (in a separate implicit parameter list), which ensures that the constraint type function evaluates to true when parameterized with the inferred type $C$ (expressed by `Constraint[C]=:=True`).

Based on the abstract generic lens type, one can use both type bounds and boolean constraints to describe the specifics of a fully generic lens. The following listing shows the definition of a generic *hoist*, which implements the abstract generic lens type and – in contrast to the pre-typed version of hoist – has no free type parameter.

Listing 5.12: A fully generic version of the *hoist* lens

```
1 class GenericHoist extends GenericLens[CompositeTerm, Term] {
2   type CBound = CompositeTerm // just a type alias to make the definitions below shorter
3   type Constraint[C <: CBound] = C#Length#Equals[_1] // using Nat's Equals type function
4   type A[C <: CBound] = C#Nth[_0]
5   def get[C <: CBound](c: C)(implicit check: Constraint[C]=:=True): A[C] = c.nth(_0)
6   def put[C <: CBound](a: A[C], c: C)(implicit check: Constraint[C]=:=True) = create(a)
7   def create[C <: CBound](a: A[C])(implicit check: Constraint[C]=:=True) = Term(a::HNil)
8 }
```

As can be seen in line 1, `GenericHoist` synchronizes between composite terms – which is in our term type hierarchy a generalization of any non-value term with a list of subterms – and terms of any term type. In line 3 *hoist*'s $C$-side constraint is encoded by the boolean expression `C#Length#Equals[_1]`, which says that the length of the subterms list of type $C$ must be 1, in other words, that the concrete term has exactly one child. `Length` is a type function which provides the length of a term's subterms list as a `Nat` type and is therefore only available with composite terms. Value terms have no subterms and consequently cannot be hoisted – the $C$-side type bound determines which helper type-functions can be used in a lens' type functions. As `CompositeTerm#Length` evaluates to `Nat`, we can use `Nat`'s type function `Equals`, which compares two type-level numbers and evaluates to a type-level boolean. This boolean type can then be checked for being true using the implicit type equality check in every lens functions.

### Composition and Usage of Fully Generic Lenses

To sequentially compose fully generic lenses, we must compose the type bounds, the boolean constraints, and the type function which expresses type $A$ in relation to $C$. The following listing shows parts of the definition of the sequential lens combinator `GenericComp` which puts two generic sublenses $l$ and $k$ (of types $L$ and $K$) in sequence.

Listing 5.13: The sequential composition lens combinator for fully generic lenses

```
1 class GenericComp[L <: GenericLens[LC,LA], K <: GenericLens[KC,KA], LA <: KC,
2   KC <: Term, LC <: Term, T <: Term, KA <: Term](l: L, k: K) extends GenericLens[LC,KA]{
3     type Constraint[C <: LC] = L#Constraint[C] && K#Constraint[L#A[C]]
4     type A[C <: LC] = K#A[L#A[C]]
5     def get[C <: LC](c: C)(implicit check: Constraint[C]=:=True): A[C] = k.get(l.get(c))
6     ...
```

As can be seen in line 1, the type parameters of `GenericComp` ensure that the $A$-side type bound of $L$, type $LA$, is a subtype of the $C$-side type bound of type $K$, type $KC$. The $C$-side constraint of type $L$, type $LC$, is also the $C$-side type bound of the composed lens. To compose the $C$-side constraints of the two sublenses, in line 3, their boolean type functions are composed with a logical type-level AND-operator written `&&`. Note that the constraint of $K$ cannot be applied directly on type $C$ but has to be applied to the result of computing type $A$ of lens type $L$, which is type $C$ of lens type $K$ (written `K#Constraint[L#A[C]]`, also in line 3). Finally in line 4, the type function `A[C]` of the composed lens is implemented by applying $K$'s type function to the result of $L$'s type function, similar to how the instance-level lens functions are composed (e.g., end of line 5).

The implementation of fully generic lenses and the implementation of their composition is more complex than with pre-typed lenses. The following listing shows that the increased complexity in the implementation enables easier composition and more flexibility .

Listing 5.14: Fully generic lens composition

```
1 val hoist2x = GenericComp(GenericHoist(), GenericHoist())//a generic lens that hoists 2x
2 // testing the same lens instance with different terms:
3 val str1 = hoist2x get Term(Term("str"::HNil)::HNil) //will compile, inferred to String
4 val int1 = hoist2x get Term(Term(12345::HNil)::HNil) //will compile, inferred to Int
5 val str2 = hoist2x get Term("str"::HNil) // won't compile: cannot be hoisted two times
6 val int2 = hoist2x get Term(Term(1234::"str"::HNil)::HNil) //won't compile: not 1 child
```

In line 1, we use the generic *hoist* lens and the generic sequential composition to compose a generic lens which hoists two times. We neither specify what the input nor what the output type should be. Still, the lens is completely type-checked. The test in line 3 compiles without errors because the passed term consists of two nested composite terms with exactly one child each, so that both the type bounds and the constraints hold; the output is inferred to be a string. The test in line 4 also compiles but its output is inferred to be an integer. This demonstrates how the type of a fully generic lens is not fixed but is inferred anew every time one of its functions is used. Consequently, the test in line 5 fails to compile because the composed type bound does not hold: the output of hoisting the first time is a value term, which cannot be hoisted anymore, and thus is no correct input for the second *hoist* lens, which expects a composite term. The test in

line 6 fails to compile because the composed constraint does not hold: the output term of hoisting the first time has more than one child, so that the second *hoist*'s constraint is violated. This shows how type bounds together with boolean type functions can be used effectively to describe and enforce the constraints of individual lenses.

### 5.3.5 Composing Pre-Typed Lenses With Type-Inferring Operators

A disadvantage of the fully generic approach is that mistakes in the composition of a fully generic lens can only be discovered by testing the composed lens (at compile-time) with test terms. With a pre-typed lens one can check whether the type arguments of a composed lens are inferred to the intended term types. Furthermore, specifying lens constraints as type-level boolean functions is more complicated than defining an input or output type pattern in a pre-typed lens type. Therefore, in this section, we revisit the pre-typed approach.

One of the main advantages of the fully generic approach over the pre-typed approach is that no explicit type-parameterization is needed. However, this advantage can be alleviated when type $C$ of the input term can be inferred from a domain object passed as the default argument, which many lenses need anyway. To be specific: In the *get-put-create* lens framework we adhere to, only those lenses do not need a default argument, which are *oblivious* and therefore, in *put*, do not rely on any information from the concrete term. Fully generic lenses are therefore more advantageous in a lens framework without *create*.

Nevertheless, explicit type parameterization is still a problem with the pre-typed approach because often we cannot use domain objects to infer the type-parameters from. With lenses which process intermediate terms there is no corresponding domain class, so that the term type has to be specified explicitly – this can be tedious and error-prone. Imagine, for instance, a lens which extracts several pieces of information from a source term, and then subsequent lenses rearrange these pieces so that their structure finally matches with the structure of a class in the target domain. With the previously presented pre-typed approach, the subsequent lenses would need to be type-parameterized explicitly with the potentially complicated intermediate term type of the output of the first lens. For explicit type parameterization, one needs to know what type parameters the subsequent lenses expect. Thus, knowledge of lens implementation is required for lens composition. This significantly reduces the accessability of a lens library.

In this section, we therefore augment the pre-typed approach with composition operators which require only the first lens in a chain of lenses to be type-parameterized explicitly; the rest of the chain is parameterized automatically by type inference. This way, one of the main disadvantages of the pre-typed approach is at least partly alleviated without loosing its advantages over the fully generic approach.

#### A Type-Inferring Operator for Sequential Composition

With the pre-typed version of sequential composition combinator *comp*, type $A$ of the left lens has to be the same as type $C$ of the right lens. This fact can be used to infer type $A$ of the left lens to automatically type-parameterize the right lens accordingly.

However, this does not work when passing both lenses to *comp* at once. One way to split up the composition is *currying*, i.e., implementing the two-parameter *comp* lens as a one-parameter lens which yields another one-parameter lens. Scala supports currying of functions conveniently: a function which is to be curried simply needs to define multiple parameter lists, instead of one parameter list with multiple parameters. To such curried functions one can pass only one parameter first and provide the other parameters later. However, currying with type parameters is not supported. One cannot provide just one of several type parameters. Instead, we define a composition operator whose implementation is specific to each lens type. To infer the type of the left lens, this composition operator must be defined as a method of the right lens. This way, the operator has insight about the type parameter structure of the right lens, so that the inferred type of the left lens can be used to type the right one correctly and pass both to the existing *comp* lens combinator. Fig. 5.11 visualizes the approach: each lens in a chain of sequentially composed lenses infers its type from the previous one, so that only the left-most lens needs to be type-parameterized explicitly.

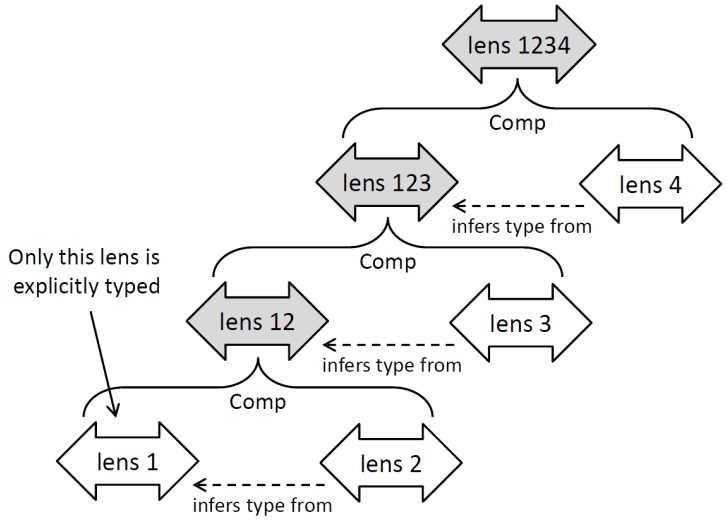



Figure 5.11: A chain of lenses composed with a type-inferring comp operator

The composition operator must be defined as a non-instance method of a lens because with instantiation the type parameters would already be set. To define non-instance methods in Scala, one defines a so-called *companion object*, which is a singleton object with the same name as the corresponding class. Methods defined in the companion object can be called in a similar manner to how static methods are called in Java. Besides supporting method names with special characters, Scala also supports the definition of right-precedence binary operators. Scala applies right-precedence by default for operators whose names end with a colon. In Listing 5.15 at line 3, we define a composition operator '`&:`' for `Hoist` that infers the abstract type of lens it is composed with (i.e., the left one). Then, a `Hoist` instance with the correct type parameter is created a passed, together with the other lens, to *comp* (also line 3).

Listing 5.15: A sequential composition operator defined in the companion object of *hoist*

```
1 class Hoist[A <: Term] extends Lens[TupleTerm[A::TNil], A] { ... } // defined previously
2 object Hoist { // the companion object of class Hoist
3   def &:[LC <: Term, A <: Term](l: Lens[LC, TupleTerm[A::TNil]]) = Comp(l, Hoist[A]())
4 }
```

In Listing 5.16, we show how this composition operator is used, and compare its usage with equivalent composition without the type-inferring operator. In line 1, a *filterN* lens is parameterized and instantiated. It is conveniently typed by inferring type $C$ of the default argument needed anyway (here of type `ContactInfo`). Afterwards, in line 2, a chain of lenses is created by composing the lens instance with the lenses *hoist* and *id*. For this to work, the composition operator '`&:`' must also be implemented by the *id* lens. Note that in contrast to using the *comp* combinator directly – shown in the comment in line 3 – no explicit type parameterization is needed anymore. This makes composing lenses easier, and lens composition easier to read. Furthermore, the composition operator makes defining new lens combinators easier, as shown in line 5 with the (re-)definition of *focus* (and compared with the earlier *focus* definition).

Listing 5.16: Using the type-inferring sequential composition operator '`&:`'

```
1 val filterCI_1 = FilterN(_1, d=ContactInfo(42,"http://")) // typed via default parameter
2 val focusCI_1 = filterCI_1 &: Hoist &: Id                 // typed via left-most lens
3 // instead of: Comp(Comp(filterC1, Hoist[ValueTerm[String]]), Id[ValueTerm[String]])
4
5 def Focus[N<:Nat, C<:Term](n:N, d:C) = FilterN(n,d) &: Hoist
6 // instead of: Focus[N<:Nat, C<:Term](n:N, d:C) = Comp(FilterN(n, d), Hoist[C#Nth[N]]())
```

To make the initial typing of the left-most lens even more comfortable, we provide a type-parameterized helper operator `$[`*type*`]`, which provides a default instance of the specified class by calling its default constructor. This way, the instantiation of the *filterN* lens in line 1 of the above listing can also be written as `FilterN(_1, $[ContactInfo])`. This operator can also be used to type lenses which do not require a default type, such as *id*, because it is easier to specify a domain type class and let the corresponding term type be inferred (by looking up implicit conversions) than specifying the latter explicitly.

### Type-Safe Parallel Composition With Lens Lists

Now that we have a more comfortable way of composing lenses sequentially, we apply the same approach to parallel lens composition as well. The *fork* lens, which we presented in Sec. 5.1.1 is one way of putting lenses in parallel. Another way of parallel composition is *Focal*'s *wmap* lens combinator, which assigns each subtree its own lens. Where *fork* uses a condition to split one tree into two, *wmap* splits a tree into all its direct subtrees. Consequently, *wmap* is parameterized with a list of lenses, which must be of the same length as the child list of trees which the composed lens is intended to process. As with the sequential composition, we want the type of this composed lens to be inferred automatically and not required to be specified manually. The challenge here is to preserve the specific type parameters of pre-typed lenses in a list of lenses: When storing the lenses in a heterogeneously typed list, only the lens types but not their type parameters are

preserved. In the following listing, we show the definition of `LensList`, a data structure which wraps a list of lenses but preserves their type parameters.

Listing 5.17: Definition and usage of a type-parameter preserving lens list

```
1 class LensList[LC <: TList, LA <: TList](val list: List[Lens[_,_]]) {
2   def ::[C <: Term, A <: Term](l: Lens[C,A]) = new LensList[C :: LC, A :: LA](l :: list)
3   def nth[N <: Nat](n: N) = list(n.toInt).asInstanceOf[Lens[LC#Nth[N],LA#Nth[N]]]
4 }
5 object LLNil extends LensList[TNil, TNil](Nil) // the end of a lens list
6 // creating a lens list:
7 val lenslist = Id($[ContactInfo]) :: Focus(_1, $[ContactInfo]) :: LLNil
```

A lens list keeps track of their elements' type parameters using two type lists (line 1): one that contains the type $C$ of each lens in the list (type list $LC$), and one that contains the type $A$ of each lens in the list (type list $LA$). To obtain the type information of every lens, `LensList` provides a prepend operator '`::`' (line 2) which works similarly as standard list creation in Scala. A `Nil` type is defined in line 5, here called `LLNil` (read: lens list Nil), which marks the end of a list. The prepend operator is implemented with right-precedence, so that elements of the list can be prepended one after another starting from the end of the list. Because this prepend operator only has one parameter and always yields a new lens list, the type parameters of the prepended lens can be inferred and the new lens list can be type-parameterized accordingly by prepending the two inferred types to the type lists of the original lens list (line 2). We also provide a type-safe accessor method `nth` which type-casts an element of the wrapped list of lenses to restore lost type parameter information, using type function `Nth` of the type lists (line 4). In line 7, we demonstrate how a lens list is created by prepending parameterized `Id` and `Focus` lens instances. Now, with such type-parameter preserving lens list, we can define a *wmap* parallel composition lens combinator whose type can be inferred automatically. We just have to infer the type lists $LC$ and $LA$ of the passed lens list, because the composed lens exactly translates between a term whose subterms have the types contained in type list $LC$, and a term whose subterms have the types contained in type list $LA$. Listing 5.18 shows parts of the definition of the *wmap* lens combinator.

Listing 5.18: A *wmap* lens combinator for parallel lens composition

```
1 // a lens which applies each lens in a list of lenses to one subterm of given term
2 class WMap[LC <: TList, LA <: TList](ll: LensList[LC,LA])
3   extends Lens[TupleTerm[LC], TupleTerm[LA]] {
4     type C = TupleTerm[LC]
5     type A = TupleTerm[LA]
6     def get(c: C): A = (ll.list zip c.subterms) map (i) => i._2.get(i._1)
7     ...
8 }
```

The implementation of *wmap*'s lens functions is straightforward and therefore only shown for *get* in line 6: First, the lens list is zipped with list of subterms of the concrete term $c$, which results in a list of tuples where each tuple contains a child of $c$ and the lens which is to be applied to this child. Then, the first component of each tuple (accessed by `._1` in Scala) is passed to the *get* function of the lens which is the the second component

of each tuple. List iteration is described in a functional manner by passing an anonymous function (defined by `(..) => ..`) to the `map` method of the list of tuples.

### 5.3.6 Conclusion

In this section, we showed how lenses from the *Focal* tree lens library can be implemented as a type-safe Scala library. First, we adapted the data model of *Focal* to fit with an object-oriented modelware setting. We then showed how type-level programming can be used to specify lenses independently of the concrete domain classes to whose instance they can be applied. We explained two alternative approaches to type-parameterization of lenses: pre-typed lenses and fully generic lenses. In the previous subsection, we showed how the pre-typed approach can be improved with type-inferring operators.

Pre-typed lenses are easier to test and – in conjunction with type-inferring combinators – have only little disadvantages compared with fully generic lenses, especially when adhering to the *get-put-create* lens framework. We therefore decided to use the pre-typed approach to implement our bidirectional MTL. We successfully implemented many of *Focal*'s lenses in a similar type-safe manner to the few lenses and combinators, whose implementation we showed in the previous sections. In the next section, we show how this lens library is further adapted so that it can be used to implement bidirectional model transformations.

## 5.4 Applying Object-Tree Lenses to Model Transformation

Adapting *Focal*'s lenses to an object-oriented data model was the first step towards a bidirectional Scala MTL. Also, automatic type checking of lens composition contributes to our goal of metamodel-aware MTL tooling. However, although we already defined a reference term type for encoding non-containment references in our object tree data model, until now we only translated *Focal*'s tree lenses to our modelware setting while taking care to preserve their semantics so that we do not need to prove their invertibility properties again. Therefore, in the examples and lenses presented so far, non-containment references did not occur yet.

In this section, we show how lenses can be used for model transformation. Therefore, in the next subsection we show how to handle non-containment references. Afterwards, we present special lenses required for model transformation. We then demonstrate their usage with the help of the Families2Persons example, which we already used in the last chapter. Finally, we sketch how the internal DSL approach allows unidirectional and bidirectional transformation descriptions to be mixed.

### 5.4.1 Handling Non-Containment References

With EMF models, we have a containment tree of model elements which can contain non-containment references. In the next three subsections, we show how a model can be converted to an object tree with cross-references and vice versa, how this conversion can be done implicitly and type-safely, and how we ensure referential integrity.

### Converting Between Models And Object-Trees with Cross-References

In Sec. 5.3.2, we showed how domain objects can be converted to typed terms. In Sec. 4.3.3, in the context of unidirectional transformations, we showed how to convert an EMF model – i.e., a graph of EMF objects – to a graph of case class instances by traversing the containment hierarchy twice: first, a corresponding case class instance of each EMF object is created while recording traces of these conversions; second, non-containment references are set using the recorded traces, because now one can be sure that all case class instances have been created. We illustrated the approach in the previous chapter in Fig. 4.7 (p. 95).

To convert between models and object trees, we use a similar approach: We traverse the containment hierarchy of the model, create a constructor term for each model element, and keep a trace of each conversion. In contrast to the case class conversion algorithm, we do not need to find the root of the containment hierarchy before traversal, because with lenses, it is always the root of the model (or a submodel) which is passed to, and returned by the outermost lens. Then, whenever during traversal we encounter a non-containment reference, we create a reference term which holds, for now, a reference to the model element to which the non-containment reference we encountered is pointing – not to the corresponding constructor term. Furthermore, we add the created reference term to a global list of reference terms. After traversal is finished, we iterate over the list of reference terms and – using the implicit conversion traces we recorded – we look up which constructor term has been created from the model element to which a reference term points, and set its reference accordingly. We refer to this process as *resolving references*.

In the other direction, however, when creating models from object trees with non-containment references, resolving references is a bit more intricate. When creating domain objects from terms, we have to instantiate the corresponding domain class. However, to the constructor or factory method of a domain class, we cannot pass a reference placeholder which later gets resolved, because a reference of a certain type is expected, and not of some proxy reference type such as `RefTerm`. We solve this as follows: Whenever setting non-containment reference in a domain object is expected, we call a helper function which defers setting the reference and returns null instead. We pass to this function the constructor term which is referenced in the object tree, and pass a pointer to the setter method of the non-containment reference attribute of the created domain object. EMF enforces public setter methods for model elements, so it is guaranteed that there is always a setter method which can be accessed. The null-returning helper function creates a *deferred reference* object which holds the referenced term and the setter method, and adds it to a global list of these deferred references. Then, after object tree traversal is finished and all domain objects have been created, we resolve references by iterating this list of deferred references and again use traces – of opposite direction this time – to find out what domain object has been created from what constructor term. We then use the saved setter methods to replace nulls in domain objects with the correct references.

Fig 5.12 shows how a model (whose containment hierarchy root is marked with 'R'), which has one non-containment reference, is implicitly converted (upper left corner, gray arrows show conversion traces) to a tree of terms and how the non-containment reference

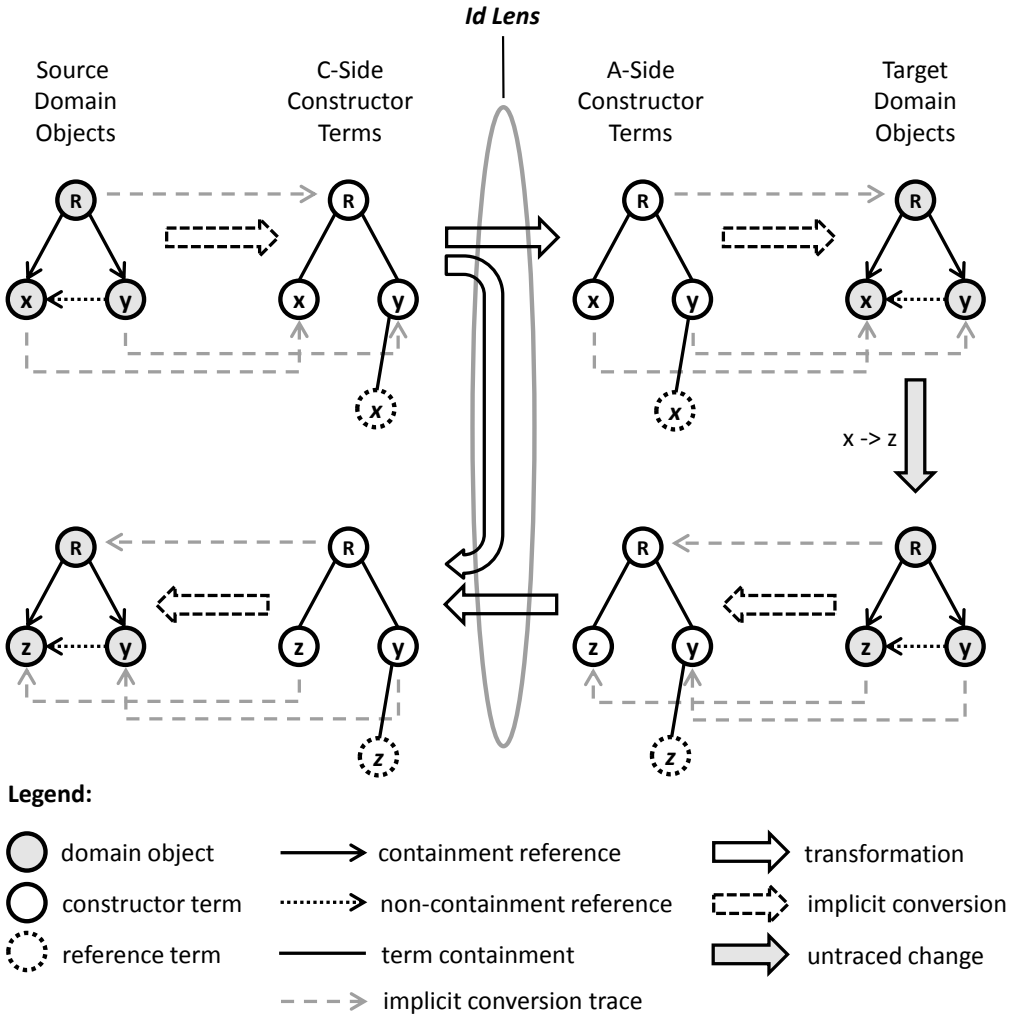



Figure 5.12: Reference handling when no references are abstracted away by *get*

becomes a reference term. This tree of terms then goes through the forward transformation *get* of an *id* lens, i.e., nothing is changed (upper row from left to right). The resulting unchanged term tree is then implicitly converted to a graph of target domain objects, i.e., the target model. This target model is then changed as node $x$ becomes node $z$. The changed target model is then implicitly converted into a term tree, and after going through the backward transformation *put* of *id*, is converted back into an updated source domain model (lower row from right to left).

### Generating Type-Safe & Reference-Safe Implicit Conversions

As with the case classes in the previous chapter, we want the conversion between models and term trees to be performed automatically when a model – more precisely the root object of the model's containment hierarchy – is passed to a lens function. Furthermore, as with the case classes, we want the required definitions of implicit conversion functions

to be generated by inspecting the involved metamodels. Recall that in contrast to the case class generation, we do not need to generate any class definitions here, because we use type-parameterizable term types instead. We only need to generate the implicit conversion functions – two functions per metamodel class. However, generating the right type annotations for those conversion functions is a bit challenging.

The following listing shows the model-to-term conversion (with slightly shortened type annotations) for the Family metamodel of the Families2Persons example (Fig. 4.1 on p. 84).

Listing 5.19: Type-safe model-to-term conversion for Family models

```
1  // generated implicit conversion from a Member domain object to a constructor term
2  implicit def member2term(m: Member):
3    CtorTerm[Member, ValueTerm[String] :: RefTerm[Family] :: RefTerm[Family]:: ... ::TNil]
4      = Traces += m -> CtorTerm(classOf[Member], m.firstName :: RefTerm(m.familyFather) ::
5       RefTerm(m.familyMother) :: RefTerm(m.familySon) :: RefTerm(m.familyDaughter) ::HNil)
6
7  // generated implicit conversion from a Family domain object to a constructor term
8  implicit def family2term[TL1 <: TList](f: Family)
9    (implicit me2t: Member => CtorTerm[Member, TL1]):
10     CtorTerm[Family, ValueTerm[String] :: CtorTerm[Member,TL1] :: ... :: TNil]
11       = Traces += f -> CtorTerm(classOf[Family], f.lastName :: f.father :: f.mother ::
12                                                  f.sons :: f.daughters :: HNil)
```

In line 3, one can see that the implicit conversion which creates a term from a `Member` domain object needs to be annotated with the return type of the accordingly type-parameterized constructor term including its complete type list (ending with `TNil`; parts of it are omitted in the listing). In lines 4–5, the constructor term is created and a trace of the conversion is stored. The syntax `x -> y` is a short form for `new Tuple2(x,y)`, so that the statement `Traces += object -> term` creates a trace-tuple and adds it to the list of traces. The children of the constructor term are passed as a heterogeneously typed list of terms (therefore ending with `HNil`), which are instantiated using the attribute values of the `Member` domain object $m$. For attribute values which are non-containment references, a reference term is created. Recall that in the Families2Persons example, a member has four back-reference fields to its containing family, of which only one is set – the one which reflects the member's role in the family – while the others are set to null.

In lines 8-12, the implicit conversion from a family object to a corresponding term is shown. Note at first, that in line 11, to create the family constructor term, we can simply pass the members of the given family object: Because a constructor term expects a list of terms, the member objects are implicitly converted to terms by the conversion function above. Thus, we do not need to explicitly traverse the model's containment tree, but simply let the implicit conversions call each other. The calls to the required conversion functions are inserted by the compiler automatically. In other words, the compiler decides how to traverse the containment tree. As a term is not created before all its children have been created, the traversal goes from the leaves of the tree to the root. So, when the root model element's corresponding term has been created, we know that for each model element a constructor term has been created.

However, in order to be able to completely specify the return type of the conversion in line 10 (again, slightly shortened), we need to infer the type list (here *TL1*) of the

constructor term type which corresponds to the member type. To be able to infer this type, in line 9, we specify explicitly in an implicit parameter list, that this function relies on an implicit conversion from members to terms. By this technique of inferring the subterms' type lists, we also prevent the type annotations of the generated conversions to become extremely long – without this, the type annotation of the conversion for the root class of a model would encode the complete structure of the term tree.

Now, in the following listing, we show the two implicit conversions for the opposite direction, i.e., for converting a term tree into a corresponding Family model.

Listing 5.20: Type-safe term-to-model conversion for Family models

```
1  // generated implicit conversion from a constructor term to a Member domain object
2  implicit def term2member
3    (t: CtorTerm[Member, ValueTerm[String] :: RefTerm[Family] :: ... :: TNil]): Member = {
4      var m: Member = null
5      m = new Member(t.nth(_0), deferref[Family](m.familyFather=_, t.nth(_1).ref), ...)
6      Traces += t -> m
7    }
8  // generated implicit conversion from a constructor term to a Family domain object
9  implicit def term2family[TL1 <: TList]
10   (t: CtorTerm[Family, ValueTerm[String] :: CtorTerm[Member, TL1] :: ... :: TNil])
11   (implicit t2me: CtorTerm[Member, TL1] => Member): Family =
12     Traces += t -> new Family(t.nth(_0), t.nth(_1), t.nth(_2), t.nth(_3), t.nth(_4))
```

In line 3, one can see that the signature of the term-to-member conversion function is exactly the opposite of its member-to-term counterpart. However, as we already mentioned, in this direction resolving references is a more intricate because we cannot simply insert placeholders for non-containment references in the first traversal. Therefore, in line 4, we first create a variable `m` for the domain object of type `Member` which we will return later, and set it to null. When creating the domain object in line 5, we need this variable for being able to refer to its own setter methods: After passing the 0th element of the term (the name) to the constructor of the `Member` class, we call inside the constructor call at the position of the first non-containment reference (omitting the other 3 for brevity) a helper method to *defer the reference* (`deferref`) and pass to it a pointer to the setter method of that reference (written as `m.familyFather=_`). In Scala, `attribute=` is the name of the setter method of an attribute, and when calling this method with an underscore, the compiler returns a function pointer with one unbound parameter – the one to set the attribute. Additionally, we pass the constructor term to which the reference term at position 1 is pointing. For now, the `deferref` method just returns null but it saves the setter method pointer and the referenced term in a list. To resolve the references, we then iterate this list and use traces to set each reference attribute accordingly.

Lines 9–12 show the term-to-family conversion, which is simpler because a family does not have any non-containment references. However, as with the family-to-term counterpart, we need to infer the type list of the member children by specifying in line 11 that we need an implicit member-to-term function. Apart from this, creating the family domain object in line 12 is simply achieved by passing the contents of the corresponding constructor term (accessed using the type-safe `nth` method) to the family class' constructor method.

**Ensuring Referential Integrity With Vertical Traces**

The presented strategy to convert between models and term trees with references only works well as long as the *get* function of a possibly composed lens does not abstract any references away (as presented in Fig 5.12), or as long as they are discarded together with the model elements which they reference. If however, the *get* function of a lens discards a reference but keeps the referenced element, this element might be changed on the abstract view side, which leads to referential corruption when propagating the change back to the concrete source side. The reason is that the *put* function of an element-discarding lens such as *filter* restores the discarded elements by looking them up in the original source-side tree, so it will restore the original references from the concrete model, which might refer to elements which have changed or have been deleted.

This problem can be solved by keeping track of what happened to updated model elements on the source side. Thus, we need a trace from a model element before it is passed to *get*, to a model element after it is returned by *put*. This is the essence of delta lenses and their vertical delta, which is a set of such vertical traces (Sec. 5.1.2). Delta lenses produce vertical traces on one side (e.g., the source side) from the vertical traces on the other side (e.g., the abstract view side). Vertical traces can, for instance, be provided by the modeling tool which is used to modify the model. However, we want our bidirectional MTL to seamlessly integrate with existing modeling tools. To allow such modeling tools to be agnostic with regards to the synchronization layer, we stick to the state-based setting where the input of a transformation is just a potentially changed model. Thus, we have no vertical view-side traces to obtain vertical source-side traces from.

Because the asymmetric state-based lens framework defines an incremental *put* function which takes the original source side model as an additional input, we can create source-side vertical traces in the *put* function of a lens: We have to connect the original source-side model element which is passed to *put* with the updated source-side element which is returned by *put*. If the same element is an input to several lenses, we simply overwrite previous traces because the trace which is recorded last is that of the outermost lens, and thus, the correct final one. We only have to keep traces from and to constructor terms because only their corresponding model elements can be referenced. Thus, we do not need to keep traces of what happened to value terms or potential intermediate terms which have no corresponding model elements.

Therefore, we use the following approach: We wrap every lens which translates between constructor terms into a bracket of two helper lenses, of which the source-side one takes care of the trace recording. The advantage of this approach is that the wrapped lens does not need to know anything about constructor terms and can simply translate between tuple terms and therefore focus on encoding structural transformation. The left helper lens *RmvCtor* removes the constructor tag of a constructor term in the *get* direction and reestablishes it in the *put* direction, and takes care of the vertical traces. The right helper lens *AddCtor* adds a constructor tag in *get* direction and removes it in the *put* direction. This approach of wrapping tuple lenses, so that they become constructor term lenses has further advantages as we will see later.

Fig. 5.13 shows a *filter* lens – parameterized to filter away every child except $x$ –

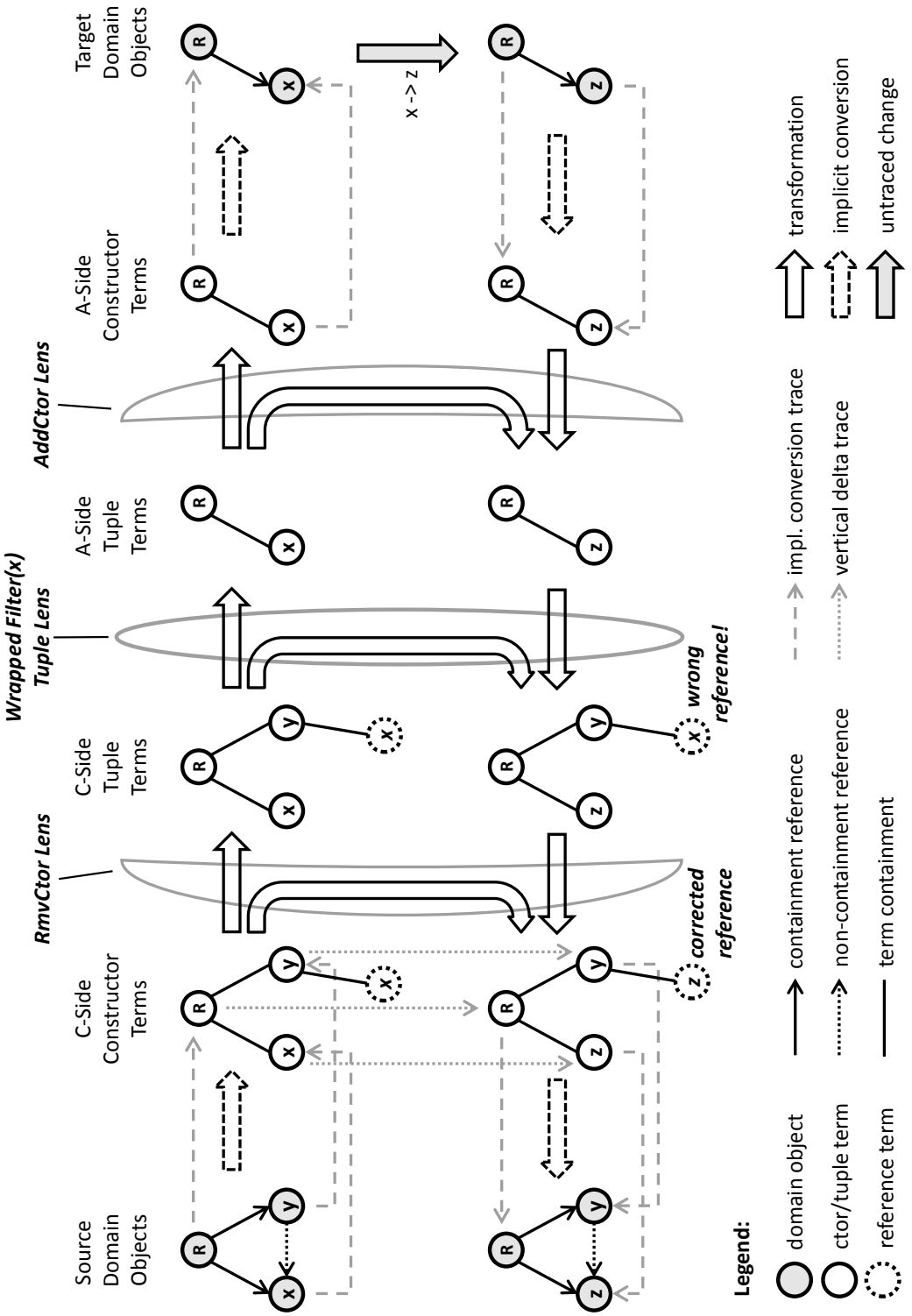



Figure 5.13: Reference handling when references are missing

which is wrapped by those two helper lenses. The term tree directly before and after *filter* consists mainly of tuple terms, whereas before *RmvCtor* and after *AddCtor*, the term tree consists mainly of constructor terms. However, apart from adding or removing constructor type tags, the two helper lenses do not change anything in the structure of the tree, so that they are structurally transparent. In other words, they are only meta-level transformations which are able to change, for instance, a class name.

As can be seen, the *filter* lens filters away child $y$, which has a non-containment reference to $x$. However $x$ is replaced by $z$ on the abstract view side (right side of the figure) and, because of the state-based lens framework, we have no trace of this change. Therefore, the *put* function of *filter* simply restores the part of the tree that was filtered away with terms of the original source side tree, including the discarded non-containment reference term which points to $x$. Thus, the restored reference term references term $x$ which has become term $z$ so that referential integrity is violated (middle of the lower row of the figure). However now, because *RmvCtor* creates vertical source-side traces (shown as finely dotted gray vertical arrows in the figure), we can resolve potentially wrong references after the tree has passed *RmvCtor*'s *put* function by looking up what has become of the referenced term and correct the wrong reference term before the whole tree is converted back to a source model with correct references (left lower corner).

### 5.4.2 Further Lenses for Model Transformation

Apart from the structurally transparent helper lenses *RmvCtor* and *AddCtor*, so far we only implemented lenses from *Focal*. Because models can be converted to object trees with reference terms, these existing tree lenses can be used for describing model transformations. In this section, we define additional lenses which are specific to our object tree data model (or have slightly different semantics) and complement *Focal*'s lenses to contribute to the practical usability of our bidirectional MTL.

#### Wrapper Lenses

In the previous section, we showed how *RmvCtor* and *AddCtor* are used to wrap a lens which is defined on tuple terms to translate between constructor terms. In the following listing, we show how these two lenses are defined type-safely.

As can be seen in line 2 and in line 11, both lenses are parameterized with the constructor tag of type `Class` which is to be removed or to be added. Furthermore, both *RmvCtor* and *AddCtor* expect an implicit conversion which converts between domain object and term or vice versa as an argument in an implicit parameter list. This has the advantage that for a lens wrapped in between those lenses, the compiler will check whether appropriate implicit conversions are in scope. For instance, if one forgot to generate the implicit conversions for the involved metamodels or if the conversions are not properly imported in the current scope, this will already cause a compile error at the time a lens instance is wrapped and not at the time the lens' functions are used.

Based on those two lenses, we use the sequential composition to provide two *wrapper* lens combinators: *WrapToCtor* composes a passed tuple lens with both *RmvCtor* and

Listing 5.21: The *RmvCtor* and the *AddCtor* lens

```
// a lens which discards the constructor tag of a CtorTerm, yielding a TupleTerm:
class RmvCtor[Ctor, TL <: TList](ctor: Class[Ctor])
  (implicit ctor2term: Ctor => CtorTerm[Ctor,TL])
  extends Lens[CtorTerm[Ctor,TL], TupleTerm[TL]] {
    type C = CtorTerm[Ctor,TL]
    type A = TupleTerm[TL]
    def get(c: C): A = TupleTerm[TL](c.subterms)
    ... // vertical trace management in the put function not shown here
}
// a lens which adds a constructor tag to a TupleTerm, yielding a CtorTerm:
class AddCtor[Ctor, TL <: TList](ctor: Class[Ctor])
  (implicit term2ctor: CtorTerm[Ctor,TL] => Ctor)
  extends Lens[TupleTerm[TL], CtorTerm[Ctor,TL]] {
    type C = TupleTerm[TL]
    type A = CtorTerm[Ctor,TL]
    def get(c: C): A = CtorTerm[Ctor,TL](c.subterms)
    ...
}
```

*AddCtor*, i.e., creating a constructor-term-to-constructor-term lens from a tuple-term-to-tuple-term lens; *WrapToValue*, expects a lens which translates between a tuple term and a value term and only composes it with *RmvCtor*, i.e., making the lens to expect a constructor term instead of a tuple term on the concrete side. For convenience, we defined a wrapper syntax in our lens DSL – `wrap(lens) as[TypeC,TypeA]` – which makes type parameterization easier and decides automatically which wrapper combinator to use, i.e., *WrapToCtor* or *WrapToValue*, depending on the wrapped lens. Furthermore, to also make direct usage of *AddCtor*, *RmvCtor*, *WrapToValue*, and *WrapToCtor* easier, we overloaded the `$[T]` operator (introduced in Sec. 5.3.5) so that it – depending on the context – either provides a default object of the specified type or the class object of the specified type. Therefore, in the following listing, where we demonstrate lens wrapping, `RmvCtor($[ContactInfo])` is a short from for `RmvCtor(classOf[ContactInfo])`.

Listing 5.22: Wrapping lenses either using *RmvCtor*/*AddCtor* or using '`wrap ... as`'

```
1 // wrapping a lens which extracts the String attribute from a ContactInfo object:
2 val filterCI_1 = FilterN(_1, $[ContactInfo])
3 val focusCI_1 = RmvCtor($[ContactInfo]) &: filterCI_1 &: Hoist
4 //  or alternatively using the wrapping syntax of our DSL
5 val focusCI_1 = wrap(filterCI_1 &: Hoist) as[Member,String]
6
7 // wrapping a lens which extracts the 0th Member attribute from a Family object:
8 val filterFam_0 = Filter(_0, $[Family])
9 val focusFam_0 = wrap(filterFam_0 &: Hoist) as[Family,Member]
```

In line 2, we first show how to manually parameterize and compose *RmvCtor* with a *filter* and a *hoist* lens for creating a *focus* lens instance which can be directly used on objects of type `ContacInfo`. In line 5, we then show how the same can be achieved, more comfortably, by using the wrapping syntax. Here, the lens is wrapped in the *WrapToValue* combinator, because the passed lens extracts a string. In line 9, we show how the wrapping syntax can be used to create a focus lens which extracts the first member of a family. In

this case, the composed lens is automatically wrapped in the *WrapToCtor* combinator because the passed lens extracts one model element from another.

Wrapping lenses this way is helpful for structuring model transformations because one can group the intermediate transformations which are needed to get from one model element to another. In other words, it is a means to finalize a chain of detailed micro-transformations and wrap it as a bidirectional version of what a rule is in a rule-based MTL. Furthermore, providing these wrapper lenses makes lens library implementation easier because defining tuple term lenses requires less complicated type annotations than defining lenses which work on constructor terms and have to take care of implicit conversions. Because the wrapping lenses are structurally transparent, and therefore behave structurally like the *id* lens, we do not need to prove the lens laws for them.

### Lenses Using Type-Annotations as Labels

An important difference between our object tree data model and the edge-labeled tree data model is that our tree nodes have a type annotation. Therefore, with our data model, we can describe lenses where this type-annotation determines the behaviour. For instance, we can define a *filter* lens which instead of a label/index, is parameterized with a type. Such a *FilterByType* lens can, for instance, filter for all integer fields of a model element. In contrast to filtering for an index, which is by definition unique, the same type-annotation can occur multiple times in one term. Therefore, the abstract type of this lens is a heterogeneous tuple term, whereas the concrete type is a homogeneous list term. Because we use heterogeneously typed lists in our tuple terms, we can find out the length of the resulting list of *get* at compile-time, and can maintain it as a `Nat` type. This is important for the *put* function because the length of the abstract side list is not allowed to change. The semantics of this lens is actually not different from *Focal*'s filter lens because we simply use the type-annotation as a different representation of what a label can be in our data model. The following listing shows parts of the definition of this lens.

Listing 5.23: The *FilterByType* lens as an example of a type-annotation-dependent lens

```
1 // a lens that filters for a certain type
2 class FilterByType[TL<:TList, T<:Term, L<:Nat](tipe: Class[T], default: TupleTerm[TL])
3   (implicit l: Length[TL,L]) //uses type-level programming to find out the result length
4   extends Lens[TupleTerm[TL], ListTerm[T]] {
5     type C = TupleTerm[TL]
6     type A = ListTerm[T]
7     def get(c: C): A = ListTerm[T](for {t <- c.subterms if t.isInstanceOf[T]} yield t)
8     ...
9 }
```

We can describe other lenses similar to *FilterByType*, that use a type-annotation as a parameter. For example, we can define a *ConditionalByType* lens or a *ForkByType* lens working that way. Furthermore, having a *FilterByType*, we can naturally also compose a *FocusByType* lens.

### Lenses for Homogeneously Typed Collection Terms

*Focal* provides a couple of lenses specifically for lists. However in *Focal*, lists are represented as nested trees with one child each, so that lenses for lists are defined recursively. Because we have special list terms (and other collection terms) in our data model, lenses for lists are described differently than in *Focal*, while being semantically equivalent. Because the children of a tree are organized as a list in our data model, defining lenses for list terms is not much different than for tuple terms. The main difference is that list terms can have an arbitrary length and must be homogeneously typed. However some lenses such as *wmap* cannot be defined for homogeneous lists, because we cannot specify one lens for each child, if we do not know the length of the list. The following listing shows *ListMap*, a lens which applies the same lens to every child of a list term. The concrete type of *ListMap* is a list term of the passed lens' concrete type; the abstract type of *ListMap* is a list term (of same length) of the passed lens' abstract type.

Listing 5.24: The *ListMap* lens as an example of a lens for homogeneously typed lists

```
1 // a lens which applies a given lens to all elements of a list
2 class ListMap[C<:Term, A<:Term](l: Lens[C,A])
3   extends Lens[ListTerm[C], ListTerm[A]] {
4     type C = ListTerm[C]
5     type A = ListTerm[A]
6     def get(c: C): A = ListTerm[T](c.subterms.map(l.get))
7     ...
8 }
```

Because the length of a homogeneous term and its inner value-structure is not known at type-level, there are less possibilities for static type-checking with lenses for homogeneous collections. For example, a *ListFilter* lens could filter away list elements on the basis of a value-based condition, so that we cannot statically analyze what the length and content of the result list will be. Also, constraints regarding the order of elements in a homogeneous term cannot be checked at compile-time. Some lenses, such as *ListReverse*, can therefore be considered value-level-only transformations as there is no change at meta-level.

### Lenses With Equality Constraints

Some lenses require to ensure value-equality on one or on both sides. For example, *Focal*'s *copy* lens copies a child of the concrete tree so that the corresponding abstract tree has two children of the same value. This way, the abstract tree is still completely determined by the concrete tree, so that the informationally asymmetric setting is still intact, although the abstract tree contains more elements that the concrete one. However, the equality of the duplicated children needs to be maintained because otherwise in the backward transformation, the two children cannot be merged without conflict. Because guaranteeing equality (or handling inequality) is often difficult, the question of whether to include lenses with equality constraints is controversially discussed in the bidirectional transformation community – the bidirectional string transformations language *Boomerang*, for instance, does not support duplication.

In general, we have the same issue, because we rely on type checking for static analysis and therefore cannot check for value equality. Nevertheless, we decided to include equality-requiring lenses in our MTL because value-based equality constraints are common in metamodeling: They are often described as OCL constraints and are a typical part of a metamodel. As our MTL is intended to describe a model synchronization layer which works below a set of modeling tools, we expect those modeling tools to check such value-based constraints when creating or modifying models. Because only intra-model equality has to be ensured, this is completely independent of heterogeneous model synchronization. Thus, we rely on modeling tools to ensure equality and do not check this in our MTL. The following listing shows a *Duplicate* lens which duplicates an index-specified child of the concrete term and prepends the duplicate to the abstract term's child list.

Listing 5.25: The *Duplicate* lens as an example of a lens which relies on value equality

```
// a lens that duplicates a child of a term and prepends it
class Duplicate[N <: Nat, TL <: TList](n: N, d: TupleTerm[TL])
  extends Lens[TupleTerm[TL], TupleTerm[TL#Nth[N] :: TL]] {
    type C = TupleTerm[TL]
    type A = TupleTerm[TL#Nth[N] :: TL] // we can only check that the type is equal
    def get(c: C): A = TupleTerm[TL#Nth[N] :: TL](c.nth(n) :: c.subterms)
    ...
}
```

A similar lens which needs equality is the *merge* lens, which requires equality of two children of the concrete tree and merges them to one child in the abstract tree.

### Tree Concatenation

In contrast to the edge-labeled tree data model, children of our terms are always organized as a list. Thus, combining two terms is achieved by concatenating their child lists. Because of different typing, we need to distinguish between tuple term concatenation and list term concatenation. Furthermore, because a lens always translates between one concrete and one abstract term, in order to concatenate two concrete terms they must be subterms of one concrete term. The following listing shows a *TupleConcat* lens which concatenates the only two subterms of one given tuple term. A slightly more advanced concatenation lens could be defined, where one can specify which two children of a term with more than two children are to be concatenated.

Listing 5.26: The *TupleConcat* lens for type-safe term concatenation

```
// a lens that concatenates two tuple terms' child lists
class TupleConcat[TL1 <: TList, TL2 <: TList](one: TupleTerm[TL1], two: TupleTerm[TL2])
  extends Lens[TupleTerm[TupleTerm[TL1]::TupleTerm[TL2]::TNil], TupleTerm[TL1:::TL2]] {
    type C = TupleTerm[TupleTerm[TL1]::TupleTerm[TL2]::TNil]
    type A = TupleTerm[TL1:::TL2]]
    def get(c: C): A = TupleTerm[TL1:::TL2](c.nth(_0).subterms ::: c.nth(_1).subterms)
    ...
}
```

### 5.4.3 A Bidirectional Version of Families2Persons

In this section we demonstrate the usage of our bidirectional Scala MTL by presenting a bidirectional version of the originally unidirectional Families2Persons example which we already used in the previous chapter to demonstrate our unidirectional Scala MTL. Therefore, we need to adapt the involved metamodels slightly, as shown in Fig. 5.14: The Family metamodel stays largely untouched – we only changed the multiplicities' lower bounds from 0 to 1 to guarantee at least one member in every field.

In the class diagrams, we explicitly show the fields of all classes. A family object consists of five fields: the family's last name and four containment references to the family's members. The family object is the root of the containment hierarchy and therefore acts as the handle of the family model. A member object consists of five fields, too: the member's first name and four back-references (i.e., non-containment references) to the family to which the member belongs. Of those four back-references, three are always null-references, and only the reference which matches the role of the member in the family is set. In the unidirectional version of the example, it is checked which of those references is not null to determine the gender of a member. We could have thought of a more convincing setup but we wanted to stay as close as possible to the original ATL example.

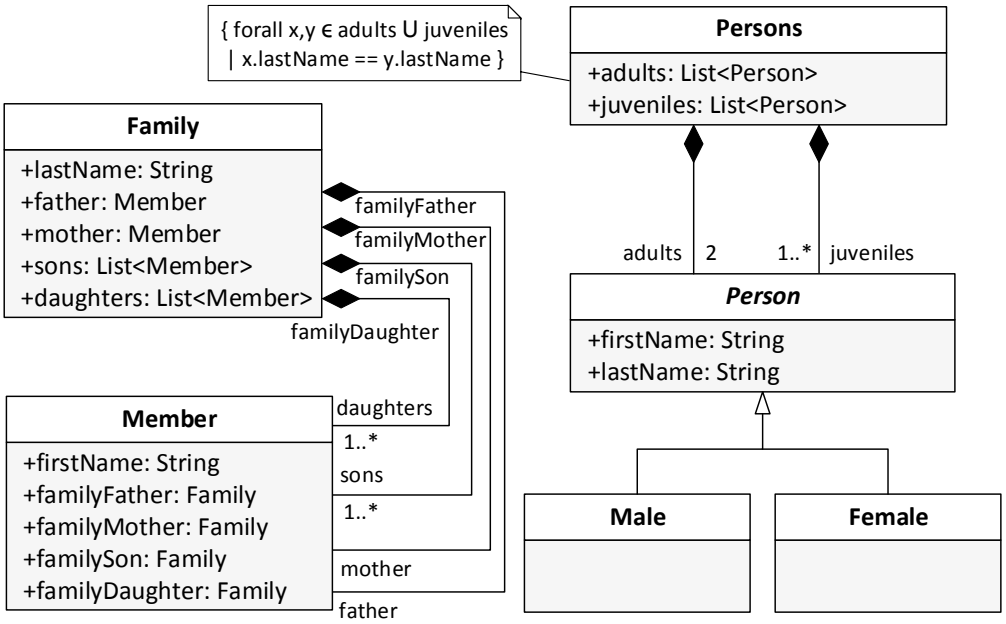



Figure 5.14: The Family and the Persons metamodel, modified for the bidirectional case

In the Persons metamodel, we add a class Persons (in addition to class Person). An object of this class is the containment hierarchy's root and contains two lists of persons: adults and juveniles. Without this root class the Persons metamodel does not fulfill our requirement that every model must have single containment hierarchy. Furthermore, the distinction between adults and juveniles allows us to implement the example without having to deal with heuristics-based name matching etc., which would distract from the

actual synchronization logic. In class Person, we represent the first name and the last name of a person as two separate fields instead of one full name string field, also to avoid cluttering the example with string analysis specifics. Importantly, we added an equality constraint which says that every person in a persons model must have the same last name. This constraint is very restrictive and might seem to render the synchronization example trivial, but it cannot be avoided when trying to stay close to the original unidirectional example in the sense that the forward transformation (which in our case is the abstracting direction) transforms from family members to persons. Because for this example, we want keep that direction of abstraction, the persons model cannot contain different last names because it would otherwise not be determined by the family model and would therefore not fit in the informationally asymmetric setting. It is of course possible to change the last name of *every* person in the persons model and propagate this change back to the family model. Furthermore, children can be added and deleted.

If we want to interpret the example in a more useful way, we can, for instance, think of the persons model as the result of a query to a bigger database of persons (with different last names) for providing only that subset (i.e., view) of persons which have the same specified last name. This way, we would describe the synchronization of one abstract view, the persons model, with another abstract view, the family model, which displays the same information differently.

### Demonstrating Lens Behaviour by Term Rewriting

In order to show how a more elaborately composed lens works, we demonstrate how a term which represents a family model is rewritten stepwise so that it finally has the structure of a persons model. In other words, we show one exemplary execution of a forward transformation. We could have used UML object diagrams instead, but found that more complex lens behaviour is best explained by term-rewriting, as this is where lenses originate from.

To make the transformation process more comprehensible, let us – for now – suppose that a family only has a father and a mother so that a family term only consists of three subterms: a value term of type String for the last name, and two constructor terms of type Member. Correspondingly, let us suppose for now that a persons term only consists of two subterms: a constructor term of type Male and a constructor term of type Female.

In the following sequence of term rewritings, we denote a constructor term by *Constructor(subterm1,subterm2,...)*, a tuple term by *(subterm1,subterm2,...)*, and a string value term by *"value"*. We denote a null-valued non-containment reference term by $\varnothing$, and a non-null non-containment reference term by $\mapsto$. In this example every non-null reference term points to the family root term. Next to the each term, we show the parameterized and/or composed lens whose forward transformation *get* (denoted by $\nearrow$) is applied to yield the next term in the rewriting sequence, i.e., the rewriting rule applied to that term. Recall that the *wmap* lens is parameterized with as many sublenses as there are subterms of the tuple term the lens is applied to, and applies each lens to one subterm.

$$\mathit{Family}(\text{“}\mathit{Simpson}\text{”}, \mathit{Member}(\text{“}\mathit{Homer}\text{”}, \mapsto, \varnothing, \varnothing, \varnothing), \mathit{Member}(\text{“}\mathit{Marge}\text{”}, \varnothing, \mapsto, \varnothing, \varnothing))$$
$$(\nearrow\textsc{RmvCtor}(\mathit{Family}))$$
$$(\text{“}\mathit{Simpson}\text{”}, \mathit{Member}(\text{“}\mathit{Homer}\text{”}, \mapsto, \varnothing, \varnothing, \varnothing), \mathit{Member}(\text{“}\mathit{Marge}\text{”}, \varnothing, \mapsto, \varnothing, \varnothing))$$
$$(\nearrow\textsc{WMap}(\textsc{Id}, \textsc{RmvCtor}(\mathit{Member}), \textsc{RmvCtor}(\mathit{Member}))$$
$$(\text{“}\mathit{Simpson}\text{”}, (\text{“}\mathit{Homer}\text{”}, \mapsto, \varnothing, \varnothing, \varnothing), (\text{“}\mathit{Marge}\text{”}, \varnothing, \mapsto, \varnothing, \varnothing))$$
$$(\nearrow\textsc{WMap}(\textsc{Id}, \textsc{Focus}(0), \textsc{Focus}(0)))$$
$$(\text{“}\mathit{Simpson}\text{”}, \text{“}\mathit{Homer}\text{”}, \text{“}\mathit{Marge}\text{”}) \qquad (\nearrow\textsc{Duplicate}(0))$$
$$(\text{“}\mathit{Simpson}\text{”}, \text{“}\mathit{Simpson}\text{”}, \text{“}\mathit{Homer}\text{”}, \text{“}\mathit{Marge}\text{”}) \qquad (\nearrow\textsc{Split}(2))$$
$$((\text{“}\mathit{Simpson}\text{”}, \text{“}\mathit{Simpson}\text{”}), (\text{“}\mathit{Homer}\text{”}, \text{“}\mathit{Marge}\text{”})) \qquad (\nearrow\textsc{Reverse})$$
$$((\text{“}\mathit{Homer}\text{”}, \text{“}\mathit{Marge}\text{”}), (\text{“}\mathit{Simpson}\text{”}, \text{“}\mathit{Simpson}\text{”})) \qquad (\nearrow\textsc{Zip})$$
$$((\text{“}\mathit{Homer}\text{”}, \text{“}\mathit{Simpson}\text{”}), (\text{“}\mathit{Marge}\text{”}, \text{“}\mathit{Simpson}\text{”}))$$
$$(\nearrow\textsc{WMap}(\textsc{AddCtor}(\mathit{Male}), \textsc{AddCtor}(\mathit{Female})))$$
$$(\mathit{Male}(\text{“}\mathit{Homer}\text{”}, \text{“}\mathit{Simpson}\text{”}), \mathit{Female}(\text{“}\mathit{Marge}\text{”}, \text{“}\mathit{Simpson}\text{”})) \qquad (\nearrow\textsc{AddCtor}(\mathit{Persons}))$$

In the first two rewriting steps, we simply remove constructor tags, i.e., we transform constructor terms to tuple terms. In the third rewriting, we extract the first name of the two member terms, discarding their non-containment references. Note the different null-reference patterns depending on the role in the family. In the backwards transformation, these references are restored by looking them up in the original family term. Next, we duplicate the last name, and by this introduce an implicit equality constraint. Next, we use a *Split* lens, which is the opposite of a tuple concatenation: parameterized with 2, the lens splits (in the forward transformation) the given term after the second subterm – a tuple term with four subterms is transformed into a tuple term with two subterms which have two subterms each. Afterwards, we reverse the order of the two terms and then zip them with each other. Finally, we add the constructor tags of the target metamodel.

### Transformation of the Children Lists

To demonstrate how the children lists are handled – which is a bit more complex – let us suppose that a family term consists only of the last name value term and two children list terms; one list with one element (the son) and one list with two elements (the daughters). We denote a list term using square brackets as in *[subterm1,subterm2,...]* and denote a list term of constructor terms by *Constructor[subterm1,subterm2,...]*. However, in the first line of the following sequence of term rewritings, we already removed the constructor tags in order to fit the term in one line. Recall that the *ListMap* lens is parameterized with one lens which is then applied to each element of a list term.

Furthermore, we use a pair of semantically inverse list-specific lenses, which we have not introduced, yet, and which belong to the group of lenses that rely on equality constraints: *Factorize* and *Distribute*. Their behaviour is inspired by the application of the distributive property in basic algebra such as in $(ax + ay) = a(x + y)$. Consequently,

both lenses are oblivious lenses, which means that in the backwards direction they do not need any information of the original concrete term as they do not discard any information in the forward transformation. *Factorize*'s *get* function takes a list term of tuple terms, where the first subterm of each tuple term is the same in every element, and produces a tuple term with two subterms: the term which was factored out, and the list with the factor removed from every element. The type of *Factorize* is `Lens[ListTerm[TupleTerm[F::TL]],TupleTerm[F::ListTerm[TupleTerm[TL]]::TNil]]`. Correspondingly, the *Distribute* lens does the opposite in its forward transformation: A given term is distributed over a given list of tuple terms, which means that it is duplicated into each tuple term and therefore is subject to an equality constraint. The type of *Distribute* therefore is `Lens[TupleTerm[F::ListTerm[TupleTerm[TL]]::TNil], ListTerm[TupleTerm[F::TL]]]`.

("Simpson", [("Bart", ∅, ∅, ↦, ∅)], [("Lisa", ∅, ∅, ∅, ↦), ("Maggie", ∅, ∅, ∅, ↦)])
(↗WMap(Id, ListMap(Split(1)), ListMap(Split1)))

("Simpson", [(("Bart"), (∅, ∅, ↦, ∅))], [(("Lisa"), (∅, ∅, ∅, ↦)), (("Maggie"), (∅, ∅, ∅, ↦))])
(↗WMap(Id, ListMap(Reverse), ListMap(Reverse))

("Simpson", [((∅, ∅, ↦, ∅), ("Bart"))], [((∅, ∅, ∅, ↦), ("Lisa")), ((∅, ∅, ∅, ↦), ("Maggie"))]
(↗WMap(Id, Factorize, Factorize)))

("Simpson", ((∅, ∅, ↦, ∅), ["Bart"]), ((∅, ∅, ∅, ↦), ["Lisa", "Maggie"]))
(↗Wmap(Id, Focus(1), Focus(1))

("Simpson", ["Bart"], ["Lisa", "Maggie"])
(↗WMap(Id, ListMap(Hoist), ListMap(Hoist))

("Simpson", ["Bart"], ["Lisa", "Maggie"]) (↗Duplicate(0))

("Simpson", "Simpson", ["Bart"], ["Lisa", "Maggie"]) (↗Split(2))

(("Simpson", "Simpson"), ([("Bart")], [("Lisa"), ("Maggie")])) (↗Zip)

(("Simpson", ["Bart"]), ("Simpson", [("Lisa"), ("Maggie")])) (↗Map(Distribute))

([("Simpson", "Bart")], [("Simpson", "Lisa"), ("Simpson", "Maggie")])
(↗Map(ListMap(Reverse)))

([("Bart", "Simpson")], [("Lisa", "Simpson"), ("Maggie", "Simpson")])
(↗WMap(ListMap(AddCtor(*Male*)),ListMap(AddCtor(*Female*))

(*Male*[("Bart", "Simpson")], *Female*[("Lisa", "Simpson"), ("Maggie", "Simpson")])
(↗SupertypeListConcat(*Person*,*Male*,*Female*))

*Person*[("Bart", "Simpson"), ("Lisa", "Simpson"), ("Maggie", "Simpson")]
(↗AddCtor(*Persons*))

The first four rewriting steps – extracting the first name of each member, as we did before, without lists – might seem more complex than necessary. If we simply apply a *ListMap(Focus(0))* to each list, we get the desired lists of first names, but then we have a problem in the backward transformation: If a child is added to the list of juveniles in

the persons model – an update whose propagation we want to allow – the *put* function of *Focus* will look for an original child representing member term to restore the discarded references. However, there is none because the element was added. Therefore, instead of simply discarding the references, we first split and reverse the member terms so we can then factor out the references before discarding them in step four. The difference is that now the *put* function of *Focus* can simply restore the references from the original – but already factorized – concrete term. This is possible because the pattern of references is always the same within one list of children because they all have the same role in the family, and thus, the references can be merged using *Factorize*.

From there on, we apply similar rewritings as in the example without lists. In rewriting step 9, we then use the *Distribute* lens to distribute the last name into each list element containing a first name. Afterwards, we add constructor tags for Male and Female terms, which in this direction is simply decided by the position of the list in the tuple term: the first list was originally the list of sons and is therefore tagged with Male; the second was originally the list of daughters and is therefore tagged with Female.

Finally, in step 12 – one step before the last one – we apply another lens which we have not introduced, yet, and which is crucial to this example: *SupertypeListConcat*. It is a lens which belongs to the group of lenses whose behaviour is determined by the terms' type annotations. In the forward direction *SupertypeListConcat* concatenates two lists terms whose types have a common supertype, so that the resulting list term has this supertype. This way, we get the desired list of adults in the Persons model – a list of type `List[Person]` and which contains both Male and Female objects (both subtypes of Person). The backward transformation is essentially a list split, just that the way how to split the list is not determined by a specified index but by checking the type annotations of each list element. Because a list term is homogeneously typed, we have no compile-time information about the individual subtype of each element in the list. We can, however, check the individual subtype at runtime and sort the elements of the list accordingly. This is possible because Scala provides means to work around the JVM's *type erasure*, i.e., the fact that the JVM does not know about type parameters at runtime. What we can check at compile-time, however, is that there are only the specified two subtypes of the specified supertype defined in the metamodel. This way, it is guaranteed that the type of each element in the list must be one of them. The type of *SupertypeListConcat* is `Lens[TupleTerm[ListTerm[SUB1],ListTerm[SUB2]], ListTerm[SUP]]`, where `SUP <: Term`, `SUB1 <: SUP`, and `SUB2 <: SUP`. Static type checking of this lens only works because we implemented our term types to be covariant to their contents.

### Constructing the Complete Families2Persons Lens

Now that we have demonstrated how the lens works, both with the parents and with the children lists, we show how the complete *Families2Persons* lens is constructed. To break down the lens definition, we first compose several sublens instances, assign names to them, and then put them together. We denote sequential lens composition with &.

$$adultName = \textsc{RmvCtor}(Member)\ \&\ \textsc{Focus}(0)$$

*childNames* =
  ListMap(Split(1) & Reverse) & Factorize & Focus(1) & ListMap(Hoist)

*distribute* =
  Split(1) & TupleDistribute & WMap(Id,Id,Distribute,Distribute)

*reverse* = Wmap(Reverse, Reverse, ListMap(Reverse), ListMap(Reverse))

*addCtors* =
  Wmap(AddCtor(*Male*), AddCtor(*Female*),
    ListMap(AddCtor(*Male*), ListMap(AddCtor(*Female*))

*sort* = Split(2) & Map(SupertypeListConcat(*Person*,*Male*,*Female*))

*families2persons* =
  WMap(Id, *adultName*, *adultName*, *childNames*, *childNames*)
    & *distribute* & *reverse* & *addCtors* & *sort* & AddCtor(*Persons*)

Next, we show how this lens definition looks like in our bidirectional Scala MTL. Listing 5.27 shows a lens definition similar to the one above (only decomposed slightly different). Obviously, type annotations make the definition in our MTL more noisy than the clean definition using the pseudo syntax above. We could easily achieve a similarly clean DSL syntax in Scala, but not without loosing extensive type checking. As one can imagine, when constructing a lens like this or even a more complex one, type checking can be tremendously helpful because there are plenty of possibilities to make mistakes when composing many small lens combinators. Because we keep track of most types using heterogeneously typed lists, many of such mistakes will be detected at compile-time and highlighted by standard Scala tooling.

Because of the usage of a type-inferring operator for sequential lens composition, only a few lenses need to be typed explicitly. An inconvenient but unavoidable consequence of the decision to support the *create* case is apparent in line 5: The *Focus* lens requires a default concrete-side term, but this term does not correspond to any domain class (in contrast to the *Focus* lens in line 2). We therefore have to explicitly specify a default term whose structure matches the concrete type of that specific *Focus* lens instance.

Because we wrap the whole lens at the end, and also wrap those of its sublenses which process constructor terms, vertical traces on the concrete side are recorded and used automatically to maintain referential integrity. Also because of the wrapping, the lens can be applied directly to family and persons models, which are automatically converted to (and from) corresponding term object trees. The availability of all required implicit conversions is also checked automatically at compile-time when wrapping the lens.

### 5.4.4 Mixing Uni- and Bidirectional Model Transformation

Because both the unidirectional MTL which we presented in Chap. 4 and the bidirectional MTL we presented in this chapter are implemented as internal Scala DSLs, they can, in general, be mixed.

Listing 5.27: Constructing a *Families2Persons* lens in our bidirectional Scala MTL

```
1  // constructing a type-safe families2persons lens:
2  val adultName = wrap( Focus(_0, $[Member]) ) as[Member,String]
3
4  val childNames = wrap( ListMap(Split(_1, $[Member]) &: Reverse) &: Factorize &:
5       Focus( _1, Term(Term(NullRef::NullRef::NullRef:: NullRef::HNil)::List(""):::HNil) )
6       as[List[Member],List[String]]
7
8  val distribute1 = Split(_1, $[Family]) &: TupleDistribute
9
10 val distribute2 = WMap(Id[String] :: Id[String] :: Distribute[String,String]) ::
11      Distribute[String,String]) :: LLNil)
12
13 val strrev = Reverse[String::String::TNil]
14
15 val reverse = WMap(strrev :: strrev :: ListMap(strrev) :: ListMap(strrev) :: LLNil)
16
17 val addCtors = WMap(AddCtor($[Male])) :: AddCtor($[Female]) :: ListMap(AddCtor($[Male]))
18      :: ListMap(AddCtor($[Female]) :: LLNil) &: Split(_2)
19
20 val sort = Map( SupertypeListConcat($[Person],$[Male],$[Female]) )
21
22 val extractNames = WMap( Id[String] :: adultName :: adultName :: childNames ::
23      childNames :: LLNil )
24
25 val rearrange = extractNames &: distribute1 &: distribute2 &: addCtors &: sort
26
27 val families2persons = wrap(rearrange) as[Family,Persons]
```

There are two ways to mix uni- and bidirectional transformation description. A unidirectional transformation can be created from a bidirectional transformation, or a bidirectional transformation can be created from two unidirectional transformations. The first direction is less interesting but easy to accomplish. The forward transformation component *get* of any lens which translates between constructor terms can be used as a rule in a unidirectional transformation. Because a rule in our ATL-like MTL mainly consists of an anonymously defined function, we can as well create a rule by directly passing the *get* function of a lens to the rule's perform method as shown in line 3 of the following listing.

Listing 5.28: Creating a unidirectional transformation rule from a lens

```
1  import families2persons.FamiliesMM._, PersonsMM._
2  val f2pLens = wrap(rearrange) as[FamilyCC,PersonsCC] //from previous f2p-lens example
3  val f2pRule = new Rule[FamilyCC, PersonsCC] perform(f2pLens.get)
```

For the unidirectional transformation language and the bidirectional transformation language to be interoperable, the implicit conversions between domain objects and term objects need to be defined on the case class types we already generate for our unidirectional transformations. Thus, domain objects are first converted to their case class counterparts and these case class objects are then converted to corresponding constructor term objects. In fact, this makes the to-term conversions easier because the to-case-class conversions already implement the flattening of inheritance hierarchies, which is also needed for term objects. The only thing we need to provide for the above list-

ing to work is a generic implicit conversion from `Function1[CtorTerm[C,TListC], CtorTerm[A,TListA]]` (the general type of the *get* function of a lens which translates between constructor terms) to `Function1[C,A]` by using available implicit conversions.

The other direction, creating a lens from two unidirectional transformations, is more interesting. However, because the backward transformation *put* of a lens is incremental, we have to extend our unidirectional MTL with discrete incremental transformation execution. In fact, the standard rule definition syntax in our unidirectional MTL is already suited for incremental transformation. The anonymously defined function takes both the source and the target model element as arguments and returns nothing – it is actually a procedure. When executing the transformation, the transformation engine creates an empty target model element using an EMF factory. This element is then passed as the second argument to the rule procedure which realizes the actual transformation logic by mutating the state of the provided target element. In an incremental transformation execution, the target model element is simply taken from the existing target model which is to be updated incrementally. Thus, we only have to provide such incremental rule execution as an alternative `transformIncr` method of our unidirectional transformation class.

For providing the possibility to mix unidirectional and bidirectional transformation definition, we then provide a lens helper method which allows an asymmetric lens to be created directly from its two components: a unary, non-incremental forward function and a binary, incremental backward function. Of course, a lens created this way possibly violates the lens laws, so it is not guaranteed that it is a well-behaved bidirectional transformation. The helper function therefore returns a special subtype of lens called `UncheckedLens`. Using this function, an unchecked lens can be created either from two transformations or from two rules because both can be represented as suitable functions.

In Listing 5.29, we first show the signature of the helper lens creation function and then create a forward and a backward transformation which are, for simplicity, realized by just one transformation rule each. Note that to define the forward transformation's rule in line 7, we use the alternative rule definition syntax with a unary function which itself creates a target model element, which is then merged with the target element created by the transformation engine. This way, the forward rule can be implicitly converted to a unary function, whereas the backward rule, which we define with the standard two-parameter syntax (line 11), can be implicitly converted to a binary function. The state-mutating procedure is converted to a side-effect free function for this. Thus, because of the availability of these conversions, an unchecked lens can be created by passing those two rules to the lens creation helper method (line 15).

Alternatively, a lens can be created from the non-incremental transformation execution method `transform` of the forward transformation and the incremental transformation execution method `transformIncr` of the backward transformation as shown in line 18. This only works because these transformation execution methods – similarly to our lens implementation – take the root element of a model's containment hierarchy as input and return the result of the transformation as the target model's root element.

Although creating lenses from two unidirectional transformations causes the maintenance and verification issues which bidirectional transformations are meant to solve (and

Listing 5.29: Creating a lens from two unidirectional transformations

```
object Lens { // a helper method for creating a lens from its component functions
  def byComponents[C<:Term,A<:Term](get: (C)=>A, put: (C,A)=>C): UncheckedLens[C,A] =...
}

// defining a forward and a backward transformation, each consisting of just one rule
val fwd = new TransformationM2M[FamiliesMM, PersonsMM]
val forwardRule = new Rule[FamilyCC,PersonsCC] perform ((fam) => { PersonsCC(...) })
fwd addRule forwardRule

val bwd = new TransformationM2M[PersonsMM, FamiliesMM]
val backwardRule = new Rule[PersonsCC,FamilyCC] perform ((pers, fam) => { fam.set... })
bwd addRule backwardRule

// creating a lens from two transformation rules
val f2pLens1 = Lens.byComponents[FamilyCC,PersonsCC](forwardRule, backwardRule)

// creating a lens from two transformations
val f2pLens2 = Lens.byComponents[FamilyCC,PersonsCC](fwd.transform, bwd.transformIncr)
```

therefore renders the lens concept useless), the option to do so – at least temporary – has a valuable advantage, especially in combination with our MTLs being internal DSLs. It allows for a soft migration from (1) model synchronizations implemented in general-purpose Java or Scala code, to (2) model synchronizations implemented with special means of a unidirectional MTL, to (3) model synchronizations implemented using the special means of a bidirectional MTL. This way, a synchronization developer can start implementation with general-purpose code in order to get the synchronization running first, and then can gradually use the means of special MTLs to improve readability, maintainability, and to take advantage of automatic verification – all without changing the development environment in the process. This can help to alleviate the problem that the limited means which special MTLs provide, often cause developers to completely fall back to their favorite programming language as soon as they do not immediately see a way how to solve a problem in a – potentially unfamiliar – special language.

## 5.5 Related Work and Conclusions

### 5.5.1 Related Work

Our approach to embed a compositional, term-rewriting-based language as an internal DSL into Scala was inspired by the work of Sloane (2008), who embedded the term-rewriting language *Stratego* into Scala as part of the *kiama*[3] project. However, Scala's type system is not used to that extent as in our approach: In *kiama*, there is just one term type, so that the internal term structure, and thus, most of the type information, is not statically kept track of. Because we use heterogeneously typed lists to keep track of a term's internal type structure, the type checking capabilities of our internal DSL exceed those of *kiama*. However, it should to be mentioned that in *Stratego generic traversal*

[3]https://code.google.com/p/kiama/

– traversing a tree with nodes of different types – is an important feature for which it is difficult to realize static type-safety[4]. Apart from *kiama*, which is only concerned with unidirectional transformation, there are several approaches to bidirectional model transformations which are not implemented as an internal DSL but are provided as stand-alone languages and tools. These external bidirectional MTLs can be divided into informationally asymmetric, symmetric, and bijective approaches. The bijective case is much easier to handle than the other two but its applications are limited – in particular, it does not meet the requirements of domain-specific workbench development.

For the asymmetric case, and in some regard similar to our approach of providing statically type-checked bidirectional transformations, Pacheco and Cunha (2010) presented a tree lens library implemented as an internal DSL in Haskell. However, their work does not aim for adapting lenses to model transformation, and therefore also provides no JVM- or EMF-integration. For asymmetric bidirectional *model* transformation, *GRoundTram*, developed by Hidaka et al. (2011), is arguably one of the most mature tools. It provides a graph-query language called *UnQL+* for specifying asymmetric bidirectional transformations. Having such a query language is a clear advantage over our approach, especially in terms of usability, because it enables relatively comfortable definition of deep graph-traversals – something which is cumbersome with our root-oriented MTL. This is partly due to the fact that *GRoundTram* is entirely graph-based – and not tree-based like our approach – and also partly because it provides less static type checking, which makes graph traversals easier. There are some attempts to integrate *GRoundTram* with EMF and with ATL, but up to now, EMF integration is still limited and not seamless.

For the symmetric case, there are mainly the OMG-standardized *QVT-Relations* MTL, and the *Triple Graph Grammar* (TGG) approach (Schürr and Klar, 2008). *QVT-Relations* seems to be significantly influenced by works on TGGs. Both *QVT-Relations* and TGGs are rule-based approaches. As we argued earlier, the QVT standard has severe issues concerning semantics of non-bijective bidirectional transformations, which might be the reason why there is currently no *QVT-Relations* tool anymore which is actively developed. TGGs have been developed for a long time and have a solid semantic foundation. However, TGG-based tools which support bidirectional model transformations only emerged recently (Leblebici et al., 2014). Because TGGs are also truly graph-based, they do not require a spanning containment tree, and are in general more expressive concerning changes of non-containment references than our lens-based MTL (Giese and Wagner, 2009). Some unidirectional TGG-based tools also provide integration with EMF (Arendt et al., 2010). However, there is still a lack of solid bidirectional TGG tools with EMF integration, which is somehow surprising taking into account the long history and amount of work put into TGGs (Golas et al., 2012).

Irrespective of informational symmetry, there are update-based approaches like *beanbag* by Xiong et al. (2009) and trace-based approaches such as the delta lenses by Diskin et al. (2011b) that we presented earlier. Because delta lenses separate delta discovery from delta propagation, they can be used to synchronize graph-based models as long as

[4]although by now, there is a Scala version of the 'scrap your boilerplate' generic traversal approach presented originally by Lämmel and Jones (2005) that makes similar use of type-level programming like our approach: https://github.com/milessabin/shapeless

a correct vertical delta of the involved models can be provided. There are also efforts towards implementing delta lenses in terms of TGGs (Hermann et al., 2011). However, update- or delta-based approaches rely on traces and are therefore harder to integrate with existing language tools which do not provide such traces. This can be alleviated by obtaining traces by heuristics-based model comparison.

Many of the aforementioned approaches to bidirectional model transformations are more powerful or allow synchronizations to be defined more comfortably. However, because none of these approaches is implemented as an internal DSL inside a general-purpose programming language, none of the presented approaches is as tool-independent as our approach: Transformation definition with code completion, transformation execution, debugging, and technological integration can all be provided by any of several available standard Scala IDE plug-ins. Therefore, one does not depend on the ongoing development and maintenance of bidirectional model transformation tooling. All that is needed, is to install a Scala tool-set and import the internal DSL library into an existing EMF- or Java-based project. Furthermore, with none of the approaches presented above is it possible to mix bidirectional transformations both with unidirectional transformations as well as with GPL code.

Because the advantages of our approach mainly stem from the internal DSL approach, we want to promote the general approach of implementing bidirectional MTLs as internal DSLs in Scala rather than the specific tree-lens based approach which we presented in this chapter. Implementing the tree-lens based, bidirectional MTL as an internal Scala DSL allowed us to demonstrate how much expressiveness and static analysis using automatic type checking can be achieved. Therefore, it would be interesting to apply the internal DSL approach also to other bidirectional transformation techniques. In the previous chapter, we already showed how Scala's implicit conversions can help to implement a rule-based MTL. This approach could possibly be applied to create, for instance, a TGG-based internal Scala MTL, as TGGs are rule-based as well.

### 5.5.2 Conclusions

In this chapter, we showed how an informationally asymmetric bidirectional MTL can be implemented as an internal DSL in Scala. Furthermore, we showed how this approach enables transferring a technique from another technological space – here, from functional programming – to the modelware technological space. We showed and explained how the tree lenses defined by *Focal* can be adapted – both conceptually and technologically – for the definition of model transformations and for integration with existing modeling technologies. The conceptual transfer relies mainly on the assumption that models are graphs which always have a spanning containment tree; an assumption which is often true with modeling technologies in general, and with EMF in particular. The technological transfer relies mainly on Scala being a JVM-language which supports both object-oriented and functional programming, and thus enables implementing functional programming techniques in an object-oriented context and allows seamless integration with Java-based technologies such as EMF.

The bidirectional MTL which we presented is therefore well-integrated with EMF:

lenses defined with our MTL can directly process the Java-based instances which constitute an EMF model at runtime. We use the Scala compiler to perform extensive static type checking using the type information which is provided by the Java classes generated from an EMF metamodel. This way, the corresponding error highlighting, syntax checks and code completion features can be provided by any Scala IDE plug-in without requiring further tooling. As a result, our MTL is tool-independent and light-weight in the sense, that it is easy to integrate with existing modeling projects and technologies.

A limitation of our approach to use the Scala compiler for transformation verification is that we can only do static analysis on things we have static type information for. We cannot ensure any value-based constraints statically. This is especially problematic for homogeneously typed list handling, where we cannot, for instance, check statically if the order of elements has changed or if elements where deleted or added. In such cases we must rely on runtime element comparison. To deal with order in lists, we could actually implement a special variant of state-based lenses called *dictionary lenses* which is specialized on this problem (Bohannon et al., 2008). Because of our self-imposed requirement to provide as much static type-safety as possible, the expressiveness of our lens-based MTL is limited to side-effect free, non-recursive transformations which work on the root of the given model. Some transformations are much more difficult to specify this way than with other means, such as a query language like *UnQL+*. It would be interesting to investigate *generating* lenses from a higher-level transformation specification, in other words, to use lenses as a kind of bidirectional byte code. We made a first approach in this direction by generating an initial lens based on model matching (Branco and Wider, 2013).

# 6 Case Study: Implementing Transformations in the NanoWorkbench

In this chapter we demonstrate that the model transformation languages (MTLs) which we developed in this dissertation can be successfully applied to the implementation of practical model transformation tasks in a domain-specific workbench. For this, we chose two exemplary transformation tasks which occur in the *NanoWorkbench* we presented in Chap. 3. In the next section, we implement a unidirectional transformation using the rule-based MTL we presented in Chap. 4. In the section thereafter, we implement a bidirectional transformation using the combinator-based MTL we presented in Chap. 5.

## 6.1 A Unidirectional Transformation for Code Generation

In the following subsections, we will first explain the transformation task, then the involved metamodels, and finally the transformation rules required to accomplish the task.

### 6.1.1 Generating Code for Multiple Targets

The *NanoDSL*, which we developed for describing experiments in the domain of simulation-driven nanostructure development, is designed for a rather imperative description of an experiment setup. The intent expressed with a typical *NanoDSL* description could be summarized like this: "First, create a slab of this size, then create a periodic lattice of holes in this slab, then delete a row of holes in the middle, and finally modify the holes around the deleted row." This workflow reflects how the domain experts approach their experiment setups and describe them to each other. As a result of tailoring the DSL to the needs of the domain experts, the *NanoDSL* enables concise description of specific types of experiments. We developed the *NanoDSL* with the concrete-syntax-first approach (see Sec. 3.1.3). The *NanoDSL*'s metamodel is automatically generated from the grammar which describes the *NanoDSL*'s concrete textual syntax. The metamodel therefore directly reflects the language's imperative character and, for instance, contains classes such as `LineDeleteOperation` whose instances constitute the model of an experiment. Automatically generating the metamodel during the development of the *NanoDSL* had several advantages. We were able to react quickly to changing requirements and were able to provide a working modeling tool to the domain experts in order to get feedback on our implementation of the requested changes.

However, the *NanoWorkbench* is integrated with simulation tools which are targets of code generation and expect a very different, rather declarative, experiment description consisting mainly of a set of geometrical objects. Individual holes are represented by

cones which have the same refraction index as the air around the slab. The slab is represented by a flat cuboid with a different refraction index. Concepts like 'lattice' or 'row of holes' do not exist and can only be represented by groups of geometrical objects. Because of the differences between how an experiment is represented on the modeling tool side and on the simulation tool side, code generation in the *NanoWorkbench* is a complex transformation. The imperative *NanoDSL* statements need to be evaluated during the transformation to create the resulting set of geometrical objects. Furthermore, the *NanoWorkbench* was developed specifically to target different simulation tools with one experiment description, so that multiple code generators were required.

When we realized that multiple simulation tools share this more declarative, geometrical description of a nanostructure, we decided to merge large parts of the different code generating model-to-text transformations into one model-to-model transformation. We identified common concepts in the different simulation tools' input descriptions and created a metamodel capturing this lowest common denominator. Because the concepts in this metamodel are mainly concerned with the geometrical structure of a nanostructure, we call it the Nanostructure metamodel, although it also defines simulation parameters. Then, instead of directly generating simulation-tool-specific code from a *NanoDSL* model, a model-to-model transformation first transforms a *NanoDSL* model into such an intermediate Nanostructure model. From this model much simpler model-to-text transformations are required to generate input code specific to the different simulation tools. Fig. 6.1 visualizes this restructuring of the *NanoWorkbench*'s code generation.

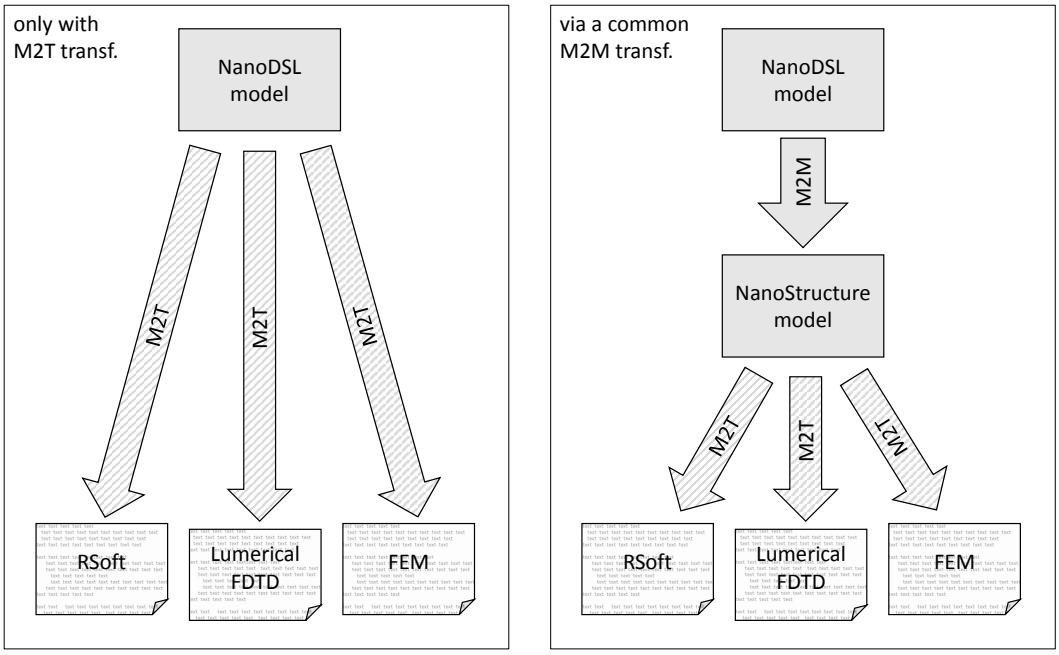



Figure 6.1: Restructuring code generation via a common model-to-model transformation

As a result of the restructuring, redundancy between the different code-generation transformations was significantly reduced. This way, more static verification of the code

generation is possible because with a model-to-model transformation – in contrast to a model-to-text transformation – it can be checked whether output elements of a transformation conform to the class types in the transformation's target metamodel.

### 6.1.2 The Involved Metamodels

In the following two subsections we briefly present those parts of the source and target metamodel which are relevant for the transformation.

#### The Source Metamodel: Nano

The *NanoDSL*'s metamodel, which is simply named *Nano*, is generated from a grammar. Therefore, the metamodel reflects the typical tree structure of a context-free grammar: Almost every grammar rule is reflected one-to-one by a corresponding class in the metamodel. Overall, the *NanoDSL*'s metamodel contains more than 50 classes and several enumerations. Therefore, we only present an excerpt which is shown in Fig. 6.2: A model – the root class – consists of several sections, among them the structure section (class `StructureSec`) which contains the description of a nanostructure. The two main concepts, slab and lattice, are both subclasses of the abstract class `Objects`. Capturing the *NanoDSL*'s imperative means, the `Lattice` class contains modifications to its periodic structure as operations. The abstract class `UnitOperations` is the root of a tree of subclasses which represent different groups of operations, such as move and delete, which themselves are further specialized with subclasses such as `LineDelete`.

#### The Target Metamodel: Nanostructure

As explained above, the target Nanostructure metamodel is designed to capture the common geometrical structure description which most simulation tools share. Models conforming to this metamodel also contain other simulation parameters but these parameters are mostly represented simply by key-value pairs.

Fig. 6.3 shows the geometry parts of the Nanostructure metamodel. All classes in this metamodel inherit from a common superclass `NamedElement` containing a string attribute `name`. However, the main class in the metamodel is `StructuralElement`. Every structural element has a position in three dimensional space (attributes `xPos`, `yPos`, and `zPos`), it refers to exactly one *material* which is characterized by its refraction index attribute, and can be part of a *group* of structural elements. The structural element class is specialized by a number of subclasses which represent basic geometrical objects such as cuboid, cone, and cylinder and contain attributes which are specific to each kind of an object (e.g., a cylinder is characterized by its radius and its height).

### 6.1.3 Implementing the Nano2Nanostructure Transformation

The complete *Nano2Nanostructure* transformation comprises several transformation rules and more than 300 lines of code. In the following subsections we therefore present only

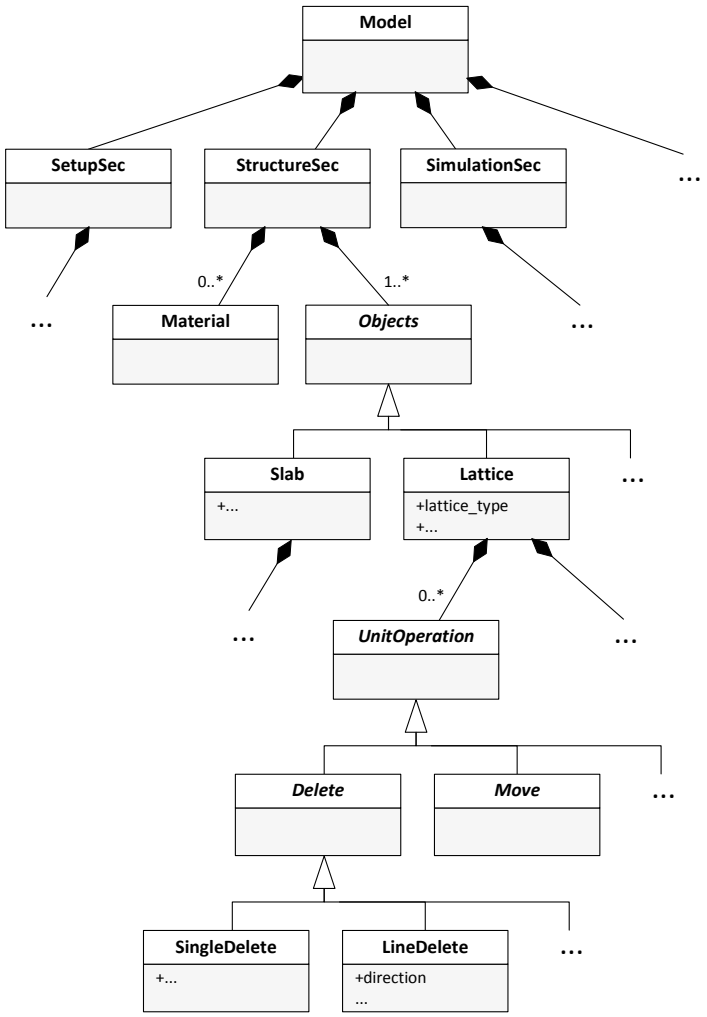



Figure 6.2: Excerpt of the NanoDSL's metamodel

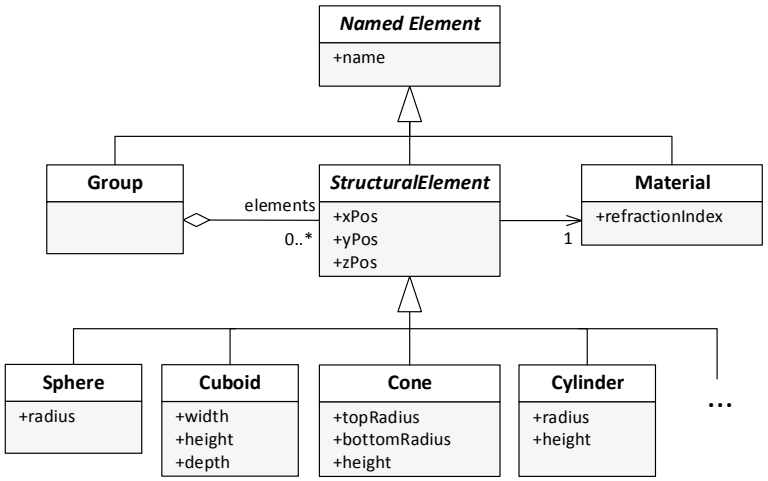



Figure 6.3: Excerpt of the Nanostructure metamodel for describing geometrical objects

some parts of the transformation which illustrate the usage and the capabilities of our unidirectional MTL.

### Creating the Transformation and Adding Two Simple Rules

The following listing shows how a model-to-model transformation is created, and how two simple transformation rules are defined and added to the transformation. First, the source and target metamodels are specified by providing appropriate type parameters to the `TransformationM2M` class' constructor. For this to work, the Scala metamodel representations which define those metamodel types (suffix `MM`) and contain the case class definitions (suffix `CC`) for each metamodel element, must have been generated (refer to Sec. 4.4). Next, a simple rule for converting material descriptions is defined (lines 4–8). In both the source and the target metamodel a material is represented by a class called `Material` and therefore the two corresponding case class types are called `MaterialCC`. In order to distinguish between the two types when type-parameterizing the rule, we need to prefix them with the namespace of the metamodel to which they belong (line 4). Because in each metamodel a material consists of a name and a refraction index, the perform-section of the *material2material* rule simply assigns the values of those attributes from the source model element *m1* to the target model element *m2*.

Listing 6.1: Creating the *Nano2Nanostructure* transformation and adding two rules to it

```
1  val nano2nanostructure = new TransformationM2M[NanoMM, NanostructureMM]
2
3  // material conversion rule
4  implicit val material2material = new Rule[NanoMM.MaterialCC, NanostructureMM.MaterialCC]
5    perform ((m1, m2) => {
6      m2.name = m1.name
7      m2.refractionIndex = m1.index.doubleValue
8    })
9
10 // rule to transform a slab to an cuboid
11 val slab2cuboid = new Rule[NanoMM.SlabCC, NanostructureMM.CuboidCC]
12   perform ((slab, cub) => {
13     cub.name = "Slab"
14     cub.xPos = slab.coordinate.x.value.doubleValue
15     cub.yPos = slab.coordinate.y.value.doubleValue
16     cub.zPos = slab.coordinate.z.value.doubleValue
17     cub.material = slab.material // implicitly applies the material2material rule above
18     cub.depth = executeCalculation(slab.thickness)
19   })
20
21 nano2nanostructure addRule material2material
22 nano2nanostructure addRule slab2cuboid
```

There is no special concept for the slab in the Nanostructure metamodel. It is represented by a cuboid named "Slab". The second transformation rule (lines 11–19) therefore converts a slab from the source metamodel to a cuboid with the same characteristics, and names it "Slab". Only the last two statements in the rule's logic are interesting. In line 17, the cuboid's material is set to the slab's material. However, the values of those material attributes have different types as there is one material class in each metamodel. Here, the

*material2material* rule defined above is implicitly applied to perform the conversion. This works because the *material2material* rule is marked with the 'implicit' keyword (line 4) and is therefore automatically used for implicit conversions. The implicit conversion is statically type-checked so that the definition of the *slab2cuboid* rule would not compile if there was no suitable rule in scope, which was marked to be implicitly available.

In line 18, the thickness of a slab – which is stored as a mathematical expression with operators, operands, etc. in the source model – needs to be calculated in order to be able to assign the result to the depth attribute of a cuboid, as this is a simple numeric value. Therefore, the whole expression stored in the slab's thickness attribute is passed to a helper function `executeCalculation` which resolves the expression and performs the calculation. The helper function is defined in general-purpose Scala code and is not shown here. In ATL, for example, such helper functions need to be defined separately in ATL's special imperative syntax. In our MTL, being embedded into Scala, we could have also implemented the calculation directly inside the rule's definition.

In lines 21–22, the two rules are added to the transformation. At this point it is statically checked that the rules which are added convert between types which belong either to the source or the target metamodel with which the transformation was type-parameterized. Of course, it is also statically checked that the output of every rule conforms to the metamodel class of the specified target model element.

### Lattice Creation and Modification

The next rule is more complex. It converts a *NanoDSL* statement which creates a lattice of holes with certain parameters to a group of cone objects which represent the resulting holes in the slab. The rule therefore creates a group model element in the nanostructure model from a lattice model element in the *NanoDSL* model. The rule implements lattice modifications by evaluating `UnitOperations` from the *NanoDSL* model and by altering the creation of cone objects accordingly. Listing 6.2 shows only those parts of the rule which are concerned with creating a hexagonal lattice and with delete-operations.

In lines 3–5 compound numerical expressions are resolved to numeric values, and from these values the parameters for the creation of cones are calculated. The group which is to be the output of the rule is named "Lattice". In line 9, we use the type-parameterizable method `getInputElement[T]` provided by our unidirectional MTL that allows us to query the set of input model elements for those of type `UnitOperation`. From this set of operations, delete operations are evaluated for obtaining a list of holes which should be omitted when creating the lattice (variable `doNotCreate`, line 13). We will show how this evaluation is implemented on p. 162.

Let's first look at the remainder of the *lattice2group* rule. Lines 13–18 show the part of the rule which handles a hexagonal lattice. In line 13, the cuboid which has been created by the *slab2cuboid* rule to represent the slab is identified by querying the set of already created model elements for an element of type `Cuboid` named "Slab". The method used for that, `getCreatedElements[T]`, is also provided by our MTL. It makes use of the fact that traces of already created model elements are recorded and made available during transformation. After modifying the slab's dimensions according to the calculated

Listing 6.2: A rule to transform from a lattice into a group of geometrical objects

```
1  val lattice2group = new Rule[NanoMM.LatticeCC, NanostructureMM.GroupCC]
2    perform ((lat, grp) => {
3      val radius = executeCalculation(lat.holeRadius)
4      val dist = executeCalculation(lat.distance)
5      val height_hexagonal_distance = Math.sqrt((dist * dist) - ((dist / 2) * (dist / 2)))
6      // ... more calculations
7      grp.name = "Lattice"
8
9      val unitOperations = getInputElements[UnitOperation]
10     val doNotCreate = resolveDeleteOperations(unitOperations)
11
12     if (lat.latticeType == LatticeType.HEXAGONAL) {
13       val slab = getCreatedElements[CuboidCC].filter(s => s.name=="Slab").first
14       slab.width = dist * maxX + height_hexagonal_distance
15       slab.height = height_hexagonal_distance * maxY
16       // ... more calculations
17       lst = generateHoles(lat.latticeType, 0-xShiftVal, 0-yShiftVal, maxX-xShiftVal,
18               maxY-yShiftVal-1, radius, dist, height_hexagonal_distance, doNotCreate)
19
20     } else if (latticeType == LatticeType.CUBICAL) { ... }
21     grp.elements.addAll(0, lst)
22   })
```

lattice parameters, the cones which represent holes are created in line 17. The previously populated list of holes which should not be created is passed (as the last argument) to the helper function which generates the holes. Finally, in line 21, all cones which have been created are added to the group which represents the lattice.

The interesting part of this transformation is the evaluation of the lattice modification operations. As indicated in Fig. 6.2, these operations are implemented in the *NanoDSL* metamodel in a multi-level subtyping hierarchy. For handling such kind of hierarchical structure, Scala's pattern matching is helpful. Listing 6.3 shows the function `resolveDeleteOperations` which is called in the *lattice2group* rule.

First, an empty set of the two-dimensional coordinates of holes to be omitted is created (line 2). Then, the set of operations is traversed, each operation is matched with different types of delete operations, and the `doNotCreate` set is updated accordingly. The first case, a single delete operation (line 6) is simple because this type of operation solely contains a 2D coordinate whose components can be added directly to the result list. However, it already illustrates how one can match into a child object, here of type `Coordinate2DCC`, bind its attributes to local variables ($x$ and $y$), and use those variables in the body of the case (after '=>'). The next case, a range delete, is slightly more complex. It contains two coordinates which specify, at least in a cubical lattice, a rectangle-shaped selection of holes by representing the rectangle's upper left and lower right corner. The calculation of which indices belong to this selection is done by the helper function `getRangeIndices` (line 9). Because delete operations can overlap, the resulting set of indices is combined with the existing `doNotCreate` set using a union set operation.

Cases 3 and 4 (lines 11–17) show how one can match not only for different types of operations but also for different attribute values. Both of the two case statements match

Listing 6.3: Use of pattern matching in the helper function for resolving delete operations

```
1  def resolveDeleteOperations(operations: Set[UnitOperation]): Set[(Int,Int)] = {
2    val doNotCreate = new HashSet[(Int,Int)]()
3
4    for(op <- operations) { // traverse the set of all operations
5      doNotCreate = let(op) match { // match an operation with different types of deletes
6        case SingleDeleteCC(Coordinate2DCC(x,y)) => doNotCreate.add((x,y))
7
8        case RangeDeleteCC(Coordinate2DCC(x1,y1), Coordinate2DCC(x2,y2))
9          => doNotCreate.union(getIndicesRange(x1, x2, y1, y2))
10
11       case LineDeleteCC(LineDirection.HORIZONTAL, _, LineSelectionType.SPAN,
12                         Coordinate2DCC(x,y), _, StepCC(stepValue))
13         => doNotCreate.union(getIndicesHorizontal(x, y, stepValue))
14
15       case LineDeleteCC(LineDirection.DIAGONALPLUS60, _, LineSelectionType.SPAN,
16                         Coordinate2DCC(x,y), _, StepCC(stepValue))
17         => doNotCreate.union(getIndicesDiagonalPlus60(x, y, stepValue))
18       // ...
19       case _ => doNotCreate
20     }
21   }
22   doNotCreate
23 }
```

for a line-delete operation but the first case statement matches for a horizontal line direction whereas the second case statement matches for a diagonal line direction. The underscores in the patterns indicate that the attribute at this position should be ignored for matching. Again helper functions are used to populate the actual set of indices which is then combined with the result set.

If no pattern matches, e.g., if the operation is not a delete-operation, the result set is to be returned unmodified. This is what the default case is specified for (line 19). The default case is also triggered if any of the accessed attributes is null-valued. This is particularly helpful because it prevents the pattern matching code to be cluttered by null-checks or by code for handling null pointer exceptions. In ATL, for instance, the same transformation would contain a lot of null-checks and if-else-constructs which would make the transformation less readable.

This example transformation shows how the availability of Scala's pattern matching helps to deal with many subtypes or variants of one general concept. This is typical for a metamodel which is generated from a grammar, where abstract grammar rules are translated to abstract types and subtyping hierarchies. An internal Scala DSL is therefore particularly suited for implementing transformations in a domain-specific workbench built with modeling-tool generating technologies such as *Xtext*. Together with the seamless technological integration which we were able to achieve because of Scala's JVM compatibility, and an expressiveness which we have shown to be similar to existing transformation languages such as ATL (see Sect. 4.5.3), we conclude that we succeeded in our goal to create a transformation language which is particularly suited for implementing model transformations in domain-specific workbenches.

## 6.2 A Bidirectional Transformation for View Synchronization

With the unidirectional transformation presented in the previous section, it is easy to create a graphical view which displays a visualization of the nanostructure. The model elements of the nanostructure model already represent geometrical objects which can simply be rendered in 2D or in 3D. We showed such a nanostructure visualization view in Fig. 3.9 in Chap. 3 (p. 53) as a part of the *NanoWorkbench*. Whenever edits are made in the textual *NanoDSL* editor and a new *NanoDSL* model is created, the unidirectional transformation is triggered and the nanostructure view is refreshed based on the transformation's output. In Fig. 3.12 on p. 56, we illustrated how the *Nano2Nanostructure* transformation is used in the *NanoWorkbench* to synchronize the structure view.

However, adding editing capabilities to such a graphical nanostructure view, i.e., allowing modifications to the nanostructure to be made directly in the visualization view and propagating those edits back to the textual editor, is difficult with the presented nanostructure model. The reason is that the nanostructure model contains the final result of creating the lattice and applying all modification operations. To propagate changes made to this declarative nanostructure model back to the imperative *NanoDSL* model, edits to geometrical objects have to be mapped to modification operations using complicated and/or ambiguous heuristics. For example, if a hole was deleted next to a line of omitted holes, one could either add a single delete operation to the *NanoDSL* model, or could update the line delete operation which caused the omission of the other holes so that it also includes the newly deleted hole.

Such a scenario of a graphical nanostructure view with editing capabilities is easier achieved and more robust with a view model which preserves the editing operations, so that those can be modified in the view and their modifications can be synchronized with the editing operations in the *NanoDSL* model. This way, the editable graphical view does not show the result of modifications but visualizes the modifications themselves so that they can be selected and edited. However, additionally to selecting existing modifications defined in the textual *NanoDSL* model and altering them, it should also be possible to add modification operations. For instance, selecting an individual hole of the lattice and deleting it should result in a single delete operation being added. Fig. 6.4 shows such a simple graphical nanostructure editor which displays both the lattice of holes and modification operations, similarly to how we visualized them in Fig. 3.6 in Chap. 3. A line delete is visualized by a red line through the holes to be deleted, and single delete is visualized as a red cross over the hole to be deleted. The editor enables selecting, adding, removing, and editing operations by means of a right-click context menu.

### 6.2.1 A Metamodel for a Graphical Nanostructure Editor

Similar to how the textual *Xtext* editor for the *NanoDSL* could automatically be generated from the grammar which describes both the textual syntax and – because the metamodel is generated from the grammar – the abstract syntax of the NanoDSL, we also want to be able to automatically generate the code for the graphical structure editor. Therefore, the underlying model the editor works on is ideally a one-to-one representation

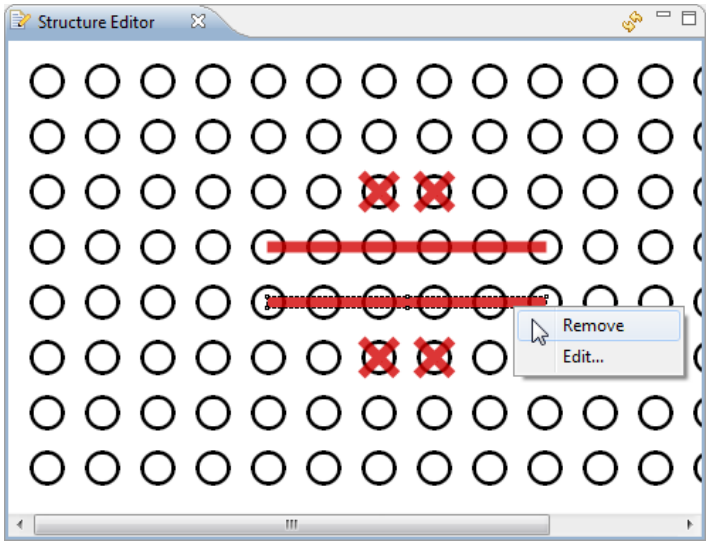



Figure 6.4: The structure editor – a graphical view which supports certain edit operations

of the GUI elements displayed by the editor, and of the user interactions they support.

Fig. 6.5 shows a simple metamodel which is designed for such a one-to-one representation of the GUI elements. The root class of the model's containment hierarchy is therefore called *GUIModel*. It contains two collections of GUI elements, objects and operations. The former collection contains 2D representations of geometrical elements of the nanostructure – such as the holes – which cannot be modified themselves. This is why the objects collection is immutable and therefore marked as read-only. The latter collection contains 2D representations of modification operations – such as a line delete – which are meant to be rendered in a layer above the geometrical objects. All elements in the two collections must provide a method *draw()* so that they can be displayed by the editor, and a method *getActions()* which provides a list of actions with which the right-click context menu of a GUI element is filled.

Class *HoleGUI*, for instance, represents the editor GUI element which visualizes a round hole two-dimensionally. It contains the instance-fields *coordinate* and *radius*, based on which the *draw()* method knows how to display a hole in the editor. The *getActions()* method returns a list consisting of a single action named "Delete hole" which, when being applied, creates a single delete operation at this hole's coordinates and adds it to the model's list of operations. Most other classes in the metamodel are editor representations of the different modification operations. A single delete operation, for instance, is characterized by its coordinates and provides only one action named "Remove" which removes this single delete operation from the list of operations. A line delete operation additionally provides an action "Edit" which enables editing the operation's properties, such as its direction.

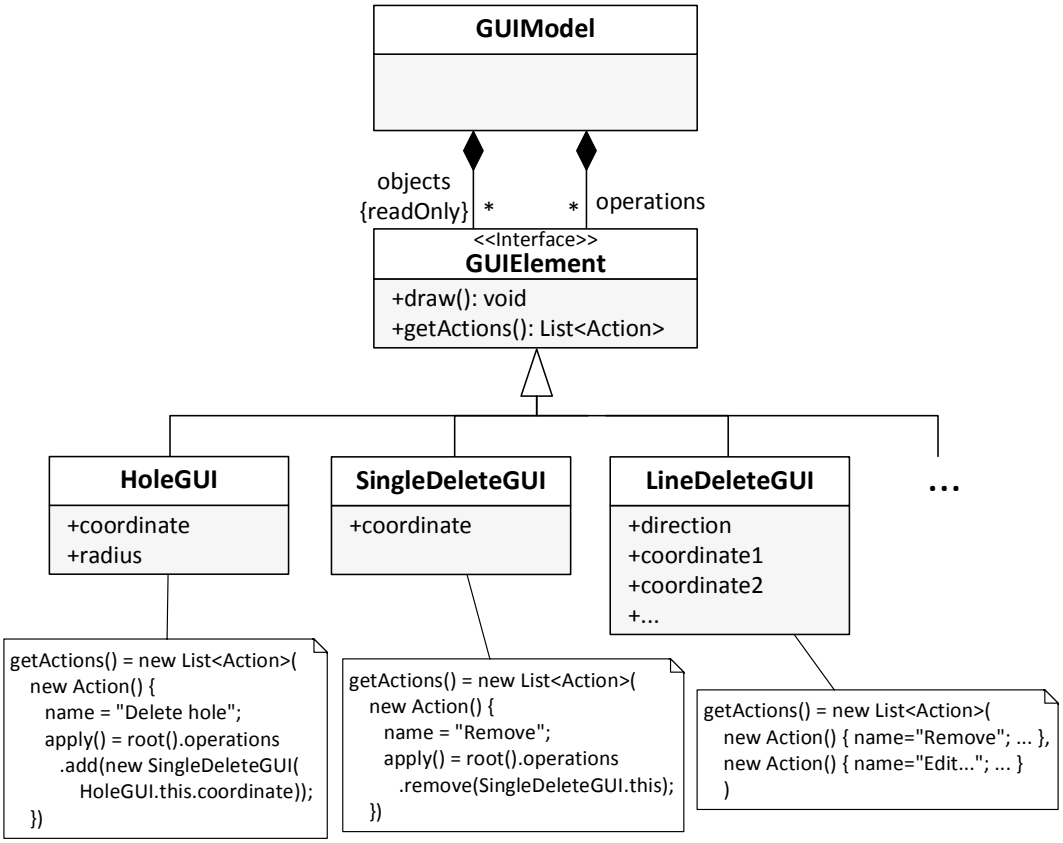



Figure 6.5: The metamodel for the underlying model of a graphical structure editor

### 6.2.2 View Synchronization Architecture and Synchronization Type

Fig. 6.6 shows how the graphical structure editor is synchronized with the textual *Xtext* editor by synchronizing their underlying models with a bidirectional model transformation. On both sides, there are trivial one-to-one mappings between the actual elements displayed in the editor and its underlying model. This is why the editor logic can be automatically generated based on the metamodel to which the underlying model conforms.

The two underlying models are synchronized by executing a non-bijective bidirectional transformation. When, for instance, an action has been applied in the structure editor, its underlying model is changed. First, the graphical structure editor itself is refreshed, which means that the updated list of operations is traversed and the *draw* method is called on each element. Afterwards, the bidirectional transformation is executed, i.e., the *NanoDSL* model is updated according to the updates in the structure editor model. This causes the textual editor to be refreshed, i.e., to display its updated underlying model. When, in turn, edits have been made in the textual editor and a new updated *NanoDSL* model has been created, the bidirectional transformation is executed to update the structure editor model according to updates in the *NanoDSL* model, and the structure editor is refreshed.

The synchronization scenario, however, is not symmetric. All updates are allowed to be

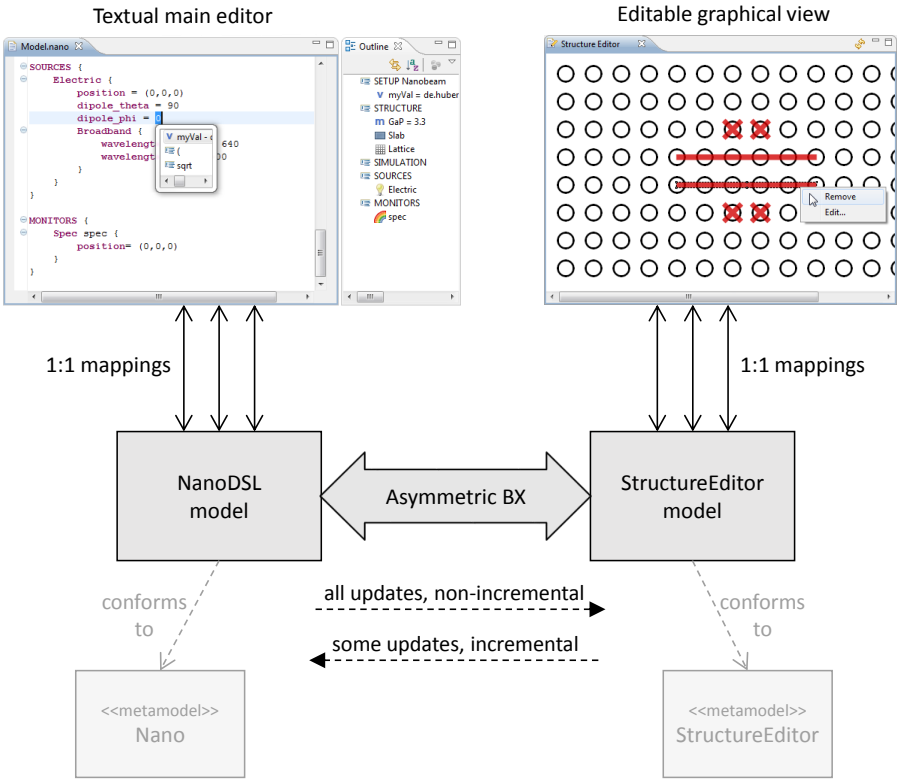



Figure 6.6: Synchronizing an editable graphical view with the main textual editor by asymmetric bidirectional model transformation

made to the *NanoDSL* model (within conformity to its metamodel) and to be propagated to the structure editor model. For the latter, however, only a limited set of updates can be applied – namely those which are accessible using the right-click context menu – and can therefore be propagated back to the *NanoDSL* model. In terms of the taxonomy which we presented in section Sec. 3.2 of Chap. 3, this means that there is organizational semi-symmetry with the structure editor model being semi-dominated organizationally. The structure editor model is also dominated informationally: there is no more than one structure editor model which corresponds to the same *NanoDSL* model, whereas there are many *NanoDSL* models which correspond to one structure editor model – namely all those *NanoDSL* models which share the same nanostructure description. Finally there is incrementality, although just state-based and only in one direction: The structure editor model gets completely regenerated every time as the result of extracting and transforming information from the *NanoDSL* model. Thus, this direction of update propagation is non-incremental. In the other direction, the original *NanoDSL* model has to be taken into account when updating it according to updates in the structure editor model because the structure-irrelevant information which has been filtered away needs to be restored. Thus, this direction is incremental, although not delta-based because it is only the original state of the *NanoDSL* model and the updated state of the structure editor model which are taken as input for the update propagation, and not the updates themselves. Such a delta-

propagation is complicated to achieve in this particular technological scenario because an *Xtext* editor always creates a completely new underlying model when something is changed in the editor, and does not provide information about individual updates.

Summing up, the synchronization type of the presented view synchronization scenario is $(\frac{1}{2}0\frac{1}{2})^{-}_{\not\Delta}$ with $\mathit{NanoDSL} \geq_{\mathsf{org}} \mathit{StructureEditor}$, $\mathit{NanoDSL} >_{\mathsf{inf}} \mathit{StructureEditor}$, and $\mathit{NanoDSL} <_{\mathsf{inc}} \mathit{StructureEditor}$. This synchronization type is supported by the computational framework of state-based lenses, and therefore by our lens-based bidirectional MTL. Because the structure editor model is dominated by the *NanoDSL* model informationally and organizationally, we call the textual editor the main editor (it displays all information and allows all edits) and call the graphical structure editor an editable view (it only displays some information and only allows some edits).

### 6.2.3 Composing a Lens for Synchronizing the Structure Editor

In the following subsections, we explain the composition of a lens which implements the synchronization between a *NanoDSL* model and a structure editor model, using the bidirectional MTL we developed in Chap. 5. Because the root element of a *NanoDSL* model is an instance of class *Model* and the root element of a structure editor model is an instance of class *GUIModel*, the lens which implements the synchronization is called *model2guimodel*. The following composition shows a simplified version of the complete lens (as before, sequential lens composition is denoted by '&'):

$$\begin{aligned} \mathit{model2guimodel} = {} & \textsc{Focus}(3) \;\&\; \textsc{Focus}(1) \;\&\; \textsc{ListFocusByType}(\mathit{Lattice}) \;\&\; \textsc{Split}(6) \\ & \&\; \textsc{WMap}(\textsc{ConstByFunction}(\mathit{generateHoles}), \textsc{Hoist} \;\&\; \textsc{ListMapByType}(\mathit{lenslist})) \end{aligned}$$

For illustrating how this lens works, on the next page, we illustrate an exemplary execution of its forward transformation in the term rewriting style which we already used in the previous chapter (we will explain the semantics and the types of its sublenses, afterwards):

The root object of a *NanoDSL* model, which is of type *Model*, mainly consists of the different sections, such as the setup section, the unit section, etc. In the first two rewriting steps, first the *Model* constructor is removed and then the fourth element, the structure section, is extracted by a *focus* lens parameterized with index 3. A structure section consists of a list of materials, characterized by a name and a refraction index, and of a list of objects, e.g., a slab and a lattice description. In the next two rewriting steps, again first the constructor is removed, this time *StructureSec*, and a *focus* lens is applied, this time for extracting the second element (index 1), which is the list of objects.

*Model*(*SetupSec*(...), *UnitSec*(...), *CodeSec*(...), *StructureSec*(...), ...)
(↗RmvCtor(*Model*))
(*SetupSec*(...), *UnitSec*(...), *CodeSec*(..), *StructureSec*(...), ...) (↗Focus(3))
*StructureSec*(*Material*[*Material*("GaP", 3.3)], *Objects*[*Slab*(...), *Lattice*(...), ...])
(↗RmvCtor(*StructureSec*))
(*Material*[*Material*("GaP", 3.3)], *Objects*[*Slab*(...), *Lattice*(...), ...]) (↗Focus(1))
*Objects*[*Slab*(...), *Lattice*(...), ...] (↗ListFocusByType(*Lattice*))
*Lattice*(hexagonal, 29, 21, 209, 0.58, 60, *UnitOperation*[*LineDelete*(...), ...])
(↗RmvCtor(*Lattice*))
(hexagonal, 29, 21, 209, 0.58, 60, *UnitOperation*[*LineDelete*(...), ...]) (↗Split(6))
((hexagonal, 29, 21, 209, 0.58, 60), (*UnitOperation*[*LineDelete*(...), ...]))
(↗WMap(ConstByFunction(*generateHoles*), Hoist))
(*GUIElement*[*HoleGUI*((1, 1), 60), ...], *UnitOperation*[*LineDelete*(...), ...])
(↗WMap(Id, ListMapByType(SingleDelete2GUI, LineDelete2GUI, ...))
(*GUIElement*[*HoleGUI*((1, 1), 60), ...], *GUIElement*[*LineDeleteGUI*(...), ...])
(↗AddCtor(*GUIModel*))

### The ListFocusByType Lens

Next, a variant of the *focus* lens is applied which we have not explained so far. The *ListFocusByType* lens is parameterized with a type and, in the forward direction, extracts an element of this type of a given homogeneously typed list of elements. Similar to the *focus* lens which works on a tuple term and is parameterized with an index, the *ListFocusByType* lens is a sequential composition of a *ListFilterByType* lens and a *ListHoist* lens. In contrast to the index-based *focus* lens where filtering for a specified index will by definition always yield a tuple term with just a single element, filtering for a type can yield multiple or no matching elements. The *ListHoist* lens however, as its tuple term counterpart, expects in the forward direction a term with exactly one child element. Therefore, the *ListFocusByType* lens has the constraint that there has to be exactly one element of the specified type in the given list term. Unfortunately, as the type system has no static type information about the specific subtypes of the elements within a homogeneously typed list, we cannot check this constraint at compile time. What we can check statically, however, is that the specified type to focus on is a subtype of the homogeneous type of the list. The type of the *ListFocusByType* lens thus can be expressed as `ListFocusByType[Super, T <: S] extends Lens[ListTerm[Super], T]`.

### The ConstByFunction Lens

After splitting the resulting lattice term at position six – which separates the fields which comprise the lattice specification from the list of lattice modification operations – the lattice specification side of the split term is sent through a special variant of the *const*

lens. The original *const* lens from *Focal* simply replaces the given concrete tree with the tree the lens is parameterized with. In the backward direction, the *const* lens expects exactly the tree it created and simply restores the original tree. If a different tree is given to the *const* lens in the backward direction, the *const* lens' typing constraint is violated. Thus, with the *const* lens a part of the abstract tree is marked immutable (with respect to synchronization) because any updates to this part violate the lens constraint and therefore cannot be propagated back to the concrete side.

Our variant of the *const* lens, *ConstByFunction*, behaves very similar to the basic *const* lens, but instead of statically specifying the constant replacement when parameterizing the lens, the constant replacement is specified by a value-level function and the replacement therefore depends on the values inside the term to be replaced. The function with which the *ConstByFunction* lens is parameterized consequently takes a term of the lens' concrete type as the single argument and returns a result of the lens' abstract type: `ConstByFunction[C <: Term, A <: Term](f: C => A) extends Lens[C, A]`.

The lens behaves like the basic *const* lens: in the backward direction it is checked that the given abstract term matches exactly the result of the specified function when applied to the original concrete term. Thus, also with the *ConstByFunction* lens, the replaced part of the abstract model cannot be updated without violating the lens constraint. It is interesting to observe that one can basically integrate a unidirectional transformation into a bidirectional transformation this way.

Now in our synchronization task at hand, the *ConstByFunction* lens is used to generate the lattice of holes from the lattice specifications. Therefore, the function it is parameterized with is called *generateHoles*. This function implements a similar logic as the *lattice2group* rule in the unidirectional transformation presented in the previous section of this chapter, but does not evaluate any lattice modification operations. Furthermore, the generated two-dimensional *HoleGUI* objects have a position and a radius only, in contrast to the *Cone* objects generated in *lattice2group* which also have a height. Immutability of the resulting set of *HoleGUI* objects in the structure editor model is reflected by the *readOnly*-attribute of the *objects* field in the *GUIModel* class.

### The ListMapByType Lens

The other side of the split lattice term, the list of lattice modification operations, is first hoisted to get the actual list term and then sent through a special kind of *ListMap* lens. The standard list *ListMap* lens applies the lens it is parameterized with to each element in the list and creates a list from the results. Each element in the concrete list is mapped to one element in the abstract list. The *ListMapByType* lens also maps every element in one list to an element in the other list but is parameterized with a list of lenses (instead of a single lens). The decision which of these lenses is applied on a particular list element is done by testing a list element's specific subtype of the list's homogeneous type. This means that every lens in the lens list which *ListMapByType* is parameterized with, must translate between a subtype of the concrete list's type and a subtype of the abstract list's type. In order to be able to check this statically, we introduce a type-restricted list of lenses. If such a list of lenses is parameterized with types $A$ and $B$ then for each

lens in the list, the concrete type must be a subtype of $A$ and the abstract type must be a subtype of $B$. In our bidirectional MTL we allow the definition of such a type-restricted lens by `e1 :: e2 :: e3 :: LLNil[A,B]`. The type arguments of the lens list with which *ListMapByType* is parameterized, must match the types of the lists between which the lens is translating. The abstract and concrete type of *ListMapByType* can therefore be automatically inferred from its lens list argument and does not need to be specified explicitly. The type of *ListMapByType* can be expressed as `ListMapByType[C,A](ll: LensList[C,A]) extends Lens[ListTerm[C],ListTerm[A]]`.

In our synchronization task at hand, *ListMapByType* is parameterized with a list of lenses which translate between the different types of lattice modification operations which share the supertype *UnitOperation* and their graphical editor representations which share the supertype *GUIElement*.

### Composing the Lens in our Bidirectional Scala MTL

The following listing shows the composition of the lens which synchronizes the textual *NanoDSL* editor's underlying model with that of the graphical structure editor using our bidirectional MTL. In lines 2–3 the three *focus* lenses are parameterized and sequentially

Listing 6.4: Constructing a lens for synchronization of the graphical structure editor

```
1  // extracting the lattice from the NanoDSL model
2  val focusOnLattice =
3     Focus(_3, $[Model]) :& Focus(_1, $[StructureSec]) &: ListFocusByType($[Lattice])
4
5  val rearrange = RmvCtor($[Lattice]) &: Split(_6)
6
7  val generateAndConvert =
8     WMap(ConstByFunction(generateHoles) :: (Hoist :& ListMapByType(lenslist)) :: LLNil)
9
10 // a helper function (or a unidirectional transformation) for generating the holes
11 def generateHoles(latticeSpecs: TupleTerm[LatticeType :: BigDecimal :: BigDecimal ::
12    Calculation :: BigDecimal :: Calculation :: TNil]): Set[GUIElement] = {
13   // ... generate set of holes according to lattice specification
14 }
15
16 // a supertype-restricted list of lenses
17 val lenslist = singledelete :: linedelete :: ... :: LLNil[UnitOperation,GUIElement]
18
19 // individual conversions from operations to lattice modification UI elements
20 val singledelete = RmvCtor($[SingleDelete]) :& AddCtor($[SingleDeleteGUI])
21 val linedelete = RmvCtor($[LineDelete]) :& AddCtor($[LineDeleteGUI])
22 // ... more trivial converions
23
24 // composing the final lens
25 val model2guimodel =
26    wrap(focusOnLattice &: rearrange :& generateAndConvert) as[Model,GUIModel]
```

composed. In our MTL we provide different ways to create a *focus* lens. If in addition to the index a constructor type is passed as an argument, a removal of this constructor is automatically applied prior to filtering and hoisting. Furthermore, no default term for the *create* function needs to be specified this way because it can be obtained by the implicit

conversion which corresponds to the constructor type. Furthermore, because we use the right-precedence sequential composition operator '`&:`', the second type parameter of the *ListFocusByType* lens (type *Objects*) can be inferred from the result of the left side of the composition and does not need to be specified explicitly. In line 5, we just remove the lattice constructor and split the resulting tuple term at index 6.

In line 7–8 we define that the two sides of the split term are sent through different lenses using the *WMap* lens combinator. The left side of the term is replaced by the result of the *generateHoles* function which is defined a few lines later. The right side of the term, the lattice modification operations, is first hoisted and then the list elements are mapped using the specified list of lenses which is also defined a few lines later.

In lines 11–12 the signature of the *generateHoles* function is shown. The type of its single parameter matches the type of the left side of the split term: A tuple term consisting of a lattice-type enumeration value and several decimal values and calculation statements. Its result type is a set of GUI elements. Specifically it creates a set of *HoleGUI* objects. As mentioned before, the logic of this function is similar to parts of the *lattice2group* rule shown in the previous section, which is why we omitted the function's body in the listing.

In line 17 the type-restricted lens list for the operation mapping is defined. Its lens elements are restricted to translate between (constructor terms of) unit operations and GUI elements. In lines 20–21 the definition of two of those operation conversion lenses is shown. Because most graphical editor representations have exactly the same properties as their unit operation counterparts, most of the conversion lenses are trivial and just replace the operation constructor type with the corresponding GUI element's constructor type.

Finally, in line 25–26 the complete lens is composed by sequentially composing the three sublenses which we defined before, and by wrapping the composition as a lens which translates between a constructor term of type *Model* (the root type in a *NanoDSL* model) and a constructor term of type *GUIModel* (the root type in a structure editor model).

### 6.2.4 Discussion & Limitations

The presented bidirectional transformation implements the synchronization of the graphical structure editor and the textual *NanoDSL* editor. Whereas any updates can be propagated from the textual editor to the graphical structure editor, the set of operations which can be propagated back from the graphical editor to the textual editor is limited.

In general, adding, deleting, and modification of any type of lattice modification operation is supported by the presented lens. In the graphical structure editor we presented, only single delete operations can be created and existing operations can only be modified by changing their attribute values. However, the editor could be extended with further functionality such as drag-and-drop moving and resizing of operations, moving of holes (i.e., creating move operations), or creation of any modification operation by dragging it from a side menu as it is often supported by graphical modeling tools. Obviously not supported by the presented synchronization are modifications to the lattice parameters because they are constantly replaced with the generated holes.

A limitation of the state-based view synchronization setting is the preservation of text layout – for example, indentation – in the textual editor because no information about

whitespace is stored in the underlying model of an *Xtext* editor. When changes are made in the structure editor, and an updated *NanoDSL* model is created correspondingly, the *Xtext* editor is refreshed by reading the updated model, mapping it to textual representation, and afterwards applying an auto-formatting algorithm which adds indentation etc. However, any added line breaks, comments or altered indentation will be lost after a view synchronization. In the structure editor, layout is preserved because the layout of the graphical editor elements is completely determined by their attributes, which all belong to the underlying model, and no further layout changes can be applied in the editor.

Finally, a limitation of the specific implementation of how our lenses handle list synchronization is that only *oblivious* lenses are supported for list mapping. This means that the backward function of mapping lenses must not rely on the original concrete list element but create an updated concrete list element only from information from the abstract list element. The reason is that when operations are added in the graphical structure editor, their original *NanoDSL* counterparts cannot be identified by their index in the list anymore. In the presented view synchronization example, this is no problem because the operation conversions are trivial and the graphical editor representations of operations contain all information of their *NanoDSL* counterparts. However, it is, for instance, not possible to focus only on some fields of an operation for the graphical representation. This limitation is also present in *Focal*. However, Bohannon et al. (2008) presented an extended lens approach called *dictionary lenses* for dealing with this problem. Such dictionary lenses could easily be added to our MTL because we already keep traces concerning which target model elements have been created from which source model elements. Using these traces, the index of the original concrete list element which corresponds to an updated abstract-side list element can be identified.

However, despite these limitations, up to now the bidirectional MTL we developed appears to be among the most suitable solutions for implementing a bidirectional model transformation like the one presented in this section. The few other existing bidirectional MTLs have other, sometimes more severe, limitations. *QVT-Relations* provides no clear semantics for non-bijective transformations and there is no up-to-date tool support for it anymore. On the other hand, other bidirectional languages which provide solid semantics for non-bijective transformations, such as *GRoundTram*, could not have been applied to the EMF-based technological setting without significant efforts.

# 7 Conclusions

We have already discussed related work in each contribution chapter. In the last chapter of this dissertation we summarize our contribution, take a look at the impact of our work, and discuss future work.

With this dissertation, we have proved our hypothesis which we presented Sec. 1.4.

> **Hypothesis**
> Model transformation languages which allow the implementation of non-bijective model synchronization as required in generated multi-view domain-specific workbenches built from unmodified modelware language workbench technologies can be implemented as internal DSLs.

For this, we presented a domain-specific workbench for simulation-driven development of optical nanostructures. We showed that such a domain-specific workbench can be described as a network of models connected by model transformations. We presented a taxonomy of model synchronization types, which we used to specify the requirements for suitable model transformation languages for this setting. According to these requirements, we implemented two model transformation languages as internal Scala DSLs. We presented an approach to the implementation of type-safe model transformation languages embedded into Scala. We then applied this approach to the development of a rule-based unidirectional transformation language and to the development of a compositional bidirectional transformation language. Finally, we showed in a small case study that these two model transformation languages can be applied to practical model synchronization tasks in the presented domain-specific workbench.

## 7.1 Impact

Scientific papers with the results of this dissertation have been peer-reviewed and published on several international workshops and conferences[1]:

- Wider, A.: *Lenses for View Synchronization in Metamodel-Based Multi-View Modeling.* In: Proceedings of the First Doctoral Symposium of the International Conference on Software Language Engineering (SLE'10), Eindhoven, The Netherlands, October 11, 2010, CEUR-WS.org (2010), 6 pages.
- Wider, A.: *Towards Combinators for Bidirectional Model Transformations in Scala.* In: Proceedings of the Fourth International Conference on Software Language En-

[1] A complete list of the author's publications can be found at the end of this dissertation along with the author's vita.

gineering (SLE'11), Braga, Portugal, July 3–4, 2011, Lecture Notes in Computer Science (LNCS) 6949, Springer (2011), 10 pages.

- Wider, A., Schmidt, M., Kühnlenz, F., Fischer, J.: *A Model-Driven Workbench for Simulation-Based Development of Optical Nanostructures.* In: Proceedings of the Second International Conference on Computer Modelling and Simulation (CSSim'11), Brno, Czech Republic, September 5–7, 2011, Brno University of Technology (2011), 9 pages.
- Wider, A.: *Towards Lenses for View Synchronization in Metamodel-Based Domain-Specific Workbenches.* In: Proceedings of the Third Workshop 'Methodische Entwicklung von Modellierungswerkzeugen' at INFORMATIK 2011, Berlin, Germany, October 6, 2011, GI-Edition of Lecture Notes in Informatics (LNI), Bonner Köllen (2011), 15 pages.
- George, L., Wider, A., Scheidgen, M.: *Type-Safe Model Transformation Languages as Internal DSLs in Scala.* In: Proceedings of the Fifth International Conference on Model Transformation (ICMT'12), Prague, Czech Republic, May 28–29, 2012, Lecture Notes in Computer Science (LNCS) 7307, Springer (2012), 16 pages.
- Wider, A.: *Implementing a Bidirectional Model Transformation Language as an Internal DSL in Scala.* In: Proceedings of the Third International Workshop on Bidirectional Transformations (BX'14), co-located with EDBT/ICDT 2014, Athens, Greece, March 28th, 2014, CEUR-WS.org (2014), 8 pages.
- Diskin, Z., Wider, A., Gholizadeh, H., Czarnecki, K.: *Towards a Rational Taxonomy for Increasingly Symmetric Model Synchronization.* In: Proceedings of the Seventh International Conference on Model Transformation (ICMT'14), York, United Kingdom, July 21–23, 2014, Lecture Notes in Computer Science (LNCS) 8568, Springer (2014), 16 pages.

Furthermore, the author of this dissertation identified topics for and supervised two master theses in the context of this dissertation:

- In his master thesis, Schmidt (2011) investigated the application of model-driven engineering for experimental physics and implemented large parts of the *NanoWorkbench*. The work emerged into a PhD research project about automatic refactorings for handling evolution of DSLs using model synchronization (Schmidt et al., 2013).
- In his master thesis, George (2012) investigated the development of model transformation languages as internal DSLs in Scala and implemented large parts of the unidirectional model transformation language presented in Chap. 4.

Our approach of implementing model transformation languages as internal DSLs in Scala in order to achieve integration with modelware technologies has attracted interest both in the MDE community and in the community of bidirectional transformations.

Notably, an integrated set of model transformation languages, implemented as internal Scala DSLs with seamless EMF-integration, has been presented recently, with explicit

reference to our approach (Křikava et al., 2014). The presented model transformation languages are also mentioned to be interoperable with the model transformation languages developed in this dissertation. In addition to our comparison of our transformation languages with existing transformation languages in terms of expressiveness (Sec. 4.5.3), the authors also compare their internal Scala model transformation languages in terms of performance, and conclude that performance of their languages is among the best compared with existing external model transformation languages.

Our work also lead to an invited research visit to investigate an application of our approach to the *GRoundTram* bidirectional transformation tool in order to broaden its practical applicability, complementing existing efforts to integrate *GRoundTram* with ATL (Sasano et al., 2011). First results were promising.

Finally, our approach of a type-safe implementation of lenses using heterogeneously typed lists seems to have had an influence[2] on the addition of similarly typed lenses to the publicly available Scala type-level programming library *shapeless*[3].

## 7.2 Future Work

Possibly the most important open question is whether the two assumptions of this dissertation, as presented in Sec. 1.4, are correct:

**Assumptions**

1. Using and combining modelware language workbench tools, with the help of model synchronization, is beneficial for creating multi-view domain-specific workbenches.
2. The achievable usability of model transformation languages which are implemented as internal Scala DSLs is acceptable for developers of domain-specific workbenches.

In order to show the first assumption, a rigorous assessment of the different approaches to domain-specific workbench development is needed. As domain-specific workbenches are built with language workbenches – if not developed entirely manual – the annual *language workbench challenge*[4], started in 2011, is a promising step in this direction. The challenge provides a forum to compare how the same task can be accomplished using different language workbenches (Völter et al., 2011). However, multi-view modeling has not been in the focus of the challenge so far.

In order to show the second assumption, an empirical study to assess the usability of different model transformation languages and their tooling is needed. We only showed with small examples that our unidirectional model transformation language is about as expressive as popular existing external model transformation languages such as ATL.

[2] judging from a post by the library's author Miles Sabin at the Google Groups Scala user group: https://groups.google.com/forum/#!msg/scala-user/-2ZT-Rxf-AQ/ITaacAsMdJIJ

[3] https://github.com/milessabin/shapeless

[4] http://languageworkbenches.net

However, in order to assess the overall usability of a language, more factors such as tool assistance, or background of the users of the language, have to be taken into account.

We are confident that it can be shown that developing model transformation languages as internal DSLs in a statically typed language such as Scala is a beneficial approach. We showed that the type checking of a host language can be used to provide tool support close to that of an external DSL. We gained the general impression that internal DSLs are well suited for domains within the software engineering domain – especially if the intended language users are familiar with the host language – because in this case an internal DSL can be extended or modified more quickly by the language users themselves, and restricts its users less. At the same time, providing and maintaining tool support for internal DSLs requires significantly less effort than for external DSLs. The recent success of *Gradle*, an internal DSL for build management, is an indication in support of this general impression.

In domains outside the software engineering domain, an external DSL may often be the better choice because its tooling enforces the boundaries of the DSL and can provide tailored user assistance. With the *NanoDSL* and the *NanoWorkbench*, we demonstrated that, by using language workbench technologies, the effort to develop an external DSL can be greatly reduced, up to the point where it is reasonable to develop a DSL tailored to the specific research field of one specific research group.

Both the internal DSL approach and language workbench tools allow more domain experts to benefit from special languages which enable them to concisely express their knowledge in a computer-processable way. Therefore, more research in both directions is needed. With regard to internal DSLs, it needs to be investigated how internal DSLs can be developed more systematically, for instance, with the help of reusable components or host-language-specific internal DSL patterns (Günther and Cleenewerck, 2010). It also needs to be investigated how the tool assistance of internal DSLs can be further improved. Our approach to use the host language's type checker, for instance, improves static verification and content-aware user assistance but error messages can still be difficult to understand. *Compiler-as-a-service* approaches could help to improve this (Moors et al., 2012). With regard to external DSL, evolution of DSLs and their tooling is a major challenge. Because the description of an external DSL and its tooling often consists of several heterogeneous artifacts, a bidirectional heterogeneous transformation language such as the one presented in this dissertation, could be used to describe automatic co-evolution of these interdependent artifacts. We proposed such approach in Schmidt et al. (2013).

It is debatable, whether describing model synchronizations with the help of tree-based lens combinators is the optimal solution for the problem of heterogeneous model synchronization. Always having to start the description of a transformation at the root of the model's containment tree can be cumbersome and can make the implementation of certain transformation tasks more complicated than necessary. However, currently, any contribution to this field is valuable because there is a significant lack of bidirectional languages which are practicably applicable and at the same time theoretically sound – although the general problem of (view) synchronization is omnipresent in software engineering. In general, combinator-based approaches, such as lenses, are promising as they

enable compositional reasoning which can, as we have shown, be performed by static type checking. This helps to ensure predictable and reasonable behaviour of automatic synchronization, which is crucial for bidirectional languages to gain user acceptance.

Generating statically type-checked tree lenses from a more high-level bidirectional description might be a beneficial approach (Branco and Wider, 2013). Apart from that, it would be highly interesting to implement other bidirectional transformation languages using our approach, e.g., *Triple Graph Grammars*, *GRoundTram*, or a combinator language based on delta lenses. Implementing *Focal*'s lens combinators as a type-safe internal Scala DSL allowed us to demonstrate how much static type checking – and thereby metamodel-awareness – can be achieved with an internal DSL in Scala while at the same time achieving a concise DSL syntax.

## 7.3 Final Remarks

The asymmetric view synchronization scenario of a domain-specific workbench with one main editor was intentionally chosen as a restricted setting. It allowed us to focus our work on a limited problem domain within the much broader problem domain of heterogeneous model synchronization and multi-view modeling in general (Antkiewicz and Czarnecki, 2007; Hesselund, 2009). This problem domain is so broad and intricate, that we could contribute only a small piece to this puzzle, namely a way to make existing conceptual approaches in this field more widely applicable in practice. We also showed that, for tackling heterogeneous model synchronization, it can be valuable to look outside of one's own technological space. Furthermore, we think that the presented taxonomy of synchronization types can serve as a guide for future research in this broad problem domain. Describing heterogeneous synchronizations concisely and in such a way that the resulting automatic synchronization is robust and predictable is a goal of high importance, with many applications beyond the scope of this dissertation, including, for example, database schema evolution or validation of object-relational mappings (Bernstein et al., 2013)

In the area of language workbenches, great advances have been made in the last couple of years, with now several language workbenches being developed simultaneously, using different approaches. It remains to be seen – and depends on more work being done as presented in this dissertation – if DSLs and domain-specific workbenches can gain major acceptance outside the domain of software engineering to fulfill the long-time goal of 'programming' being done by domain experts themselves.

# Bibliography

M. Antkiewicz and K. Czarnecki. Design Space of Heterogeneous Synchronization. In *Proceeding of the 2nd Summerschool on Generative and Transformational Techniques*, pages 3–46. Springer, 2007.

Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Model Driven Engineering Languages and Systems*, pages 121–135. Springer, 2010.

Colin Atkinson and Thomas Kühne. Model-Driven Development: A Metamodeling Foundation. *Software, IEEE*, 20(5):36–41, 2003.

Paolo Atzeni and Riccardo Torlone. Management of Multiple Models in an Extensible Database Design Tool. In *Advances in Database Technology - EDBT'96, 5th International Conference on Extending Database Technology, Avignon, France, March 25-29, 1996, Proceedings*, volume 1057 of *Lecture Notes in Computer Science*, pages 79–95. Springer, 1996.

Kacper Bąk, Dina Zayan, Krzysztof Czarnecki, Michał Antkiewicz, Zinovy Diskin, Andrzej Wąsowski, and Derek Rayside. Example-Driven Modeling: Model = Abstractions + Examples. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1273–1276. IEEE Press, 2013.

H. Barringer and K. Havelund. TraceContract: A Scala DSL for Trace Analysis. *FM 2011: Formal Methods*, pages 57–72, 2011.

M. Barth, J. Kouba, J. Stingl, B. Löchel, and O. Benson. Modification of visible spontaneous emission with silicon nitride photonic crystal nanocavities. *Optics Express*, 15 (25):17231–17240, 2007.

Jon Bentley. Programming Pearls: Little Languages. *Communications of the ACM*, 29 (8):711–721, 1986.

Philip A Bernstein, Marie Jacob, Jorge Pérez, Guillem Rull, and James F Terwilliger. Incremental Mapping Compilation in an Object-To-Relational Mapping System. In *Proceedings of the 2013 international conference on Management of data*, pages 1269–1280. ACM, 2013.

J. Bézivin, G. Dupé, F. Jouault, G. Pitette, and J.E. Rougui. First Experiments with the ATL Model Transformation Language: Transforming XSLT into XQuery. In *2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture*, page 50, 2003.

Jean Bézivin. On the Unification Power of Models. *Software & Systems Modeling*, 4(2): 171–188, 2005.

Jean Bézivin. Model Driven Engineering: An Emerging Technical Space. In *Generative and transformational techniques in software engineering*, pages 36–64. Springer, 2006.

E. Biermann, C. Ermel, and G. Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In *MoDELS*, pages 53–67. Springer, 2008.

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful Lenses for String Data. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, CA*, pages 407–419, January 2008.

Moisés Castelo Branco and Arif Wider. Generating Preliminary Edit Lenses from Automatic Pattern Discovery in Business Process Modeling. In Rébecca Deneckère and Henderik Alex Proper, editors, *Proceedings of the CAiSE'13 Forum at the 25th International Conference on Advanced Information Systems Engineering (CAiSE), Valencia, Spain, June 20th, 2013*, volume 998 of *CEUR Workshop Proceedings*, pages 65–72. CEUR-WS.org, 2013.

Noam Chomsky. Aspects of the Theory of Syntax. *MIT Press*, 1965.

A. Clark, P. Sammut, and J. Willans. *Applied Metamodelling: A Foundation for Language Driven Development, 2nd edition.* Ceteva, 2008.

T. Clark, A. Evans, S. Kent, and P. Sammut. The MMF approach to engineering object-oriented design languages. 2001.

J. Cuadrado, J. Molina, and M. Tortosa. RubyTL: A Practical, Extensible Transformation Language. In *MDA – Foundations and Applications*, pages 158–172. Springer, 2006.

Jesús Sánchez Cuadrado and Jesús García Molina. A Plugin-Based Language to Experiment with Model Transformation. In *MoDELS*, pages 336–350, 2006.

Jesús Sánchez Cuadrado and Jesús García Molina. Modularization of Model Transformations Through a Phasing Mechanism. *Software and Systems Modeling*, 8(3):325–345, 2009.

J.S. Cuadrado and J.G. Molina. Approaches for Model Transformation Reuse: Factorization and Composition. In *1st International Conference on Theory and Practice of Model Transformations (ICMT'08)*, pages 168–182. Springer, 2008. ISBN 978-3-540-69926-2.

Alcino Cunha, José Nuno Oliveira, and Joost Visser. Type-Safe Two-Level Data Transformation. In *FM 2006: Formal Methods*, pages 284–299. Springer, 2006.

K. Czarnecki and S. Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2010. ISSN 0018-8670.

Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and

James F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. GRACE Meeting notes, state of the art, and outlook. In *International Conference on Model Transformations (ICMT), Zurich, Switzerland*, volume 32, pages 260–283. Springer, June 2009. Invited Paper.

Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas. From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case. In *MoDELS*, pages 304–318, 2011a.

Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *Journal of Object Technology*, 10:6: 1–25, 2011b.

Zinovy Diskin, Arif Wider, Hamid Gholizadeh, and Krzysztof Czarnecki. Towards a Rational Taxonomy for Increasingly Symmetric Model Synchronization. In Davide Di Ruscio and Dániel Varró, editors, *Theory and Practice of Model Transformations - 7th International Conference, ICMT 2014, Held as Part of STAF 2014, York, UK, July 21-22, 2014. Proceedings*, volume 8568 of *Lecture Notes in Computer Science*, pages 57–73. Springer, 2014. ISBN 978-3-319-08788-7.

Sven Efftinge and Markus Völter. oAW xText: A Framework for Textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, volume 32, page 118, 2006.

H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, and G. Taentzer. Information Preserving Bidirectional Model Transformations. In *Fundamental Approaches to Software Engineering, 10th International Conference, FASE 2007*, volume 4422, pages 72–86. Springer, 2007.

Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, et al. The State of the Art in Language Workbenches. In *Software Language Engineering*, pages 197–217. Springer, 2013.

Joachim Fischer, Eckhardt Holz, Andreas Prinz, and Markus Scheidgen. Tool-Based Language Development. *Computer Networks*, 49(5):676–688, 2005.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for Bi-Directional Tree Transformations: A Linguistic Approach to the View Update Problem. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 233–246. ACM, 2005.

J.N. Foster. *Bidirectional Programming Languages*. PhD thesis, University of Pennsylvania, 2009.

J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, and A. Schmitt. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29 (3):17, 2007.

M. Fowler. Language Workbenches: The Killer-App for Domain-Specific Languages. *Accessed online from: http: // www.martinfowler.com / articles / languageWorkbench.html*, 2005. URL `http://www.martinfowler.com/articles/languageWorkbench.html`.

Martin Fowler. *Domain-Specific Languages*. Addison Wesley Signature Series. Addison-Wesley, 2010. ISBN 9780321712943.

Miguel Garcia. Bidirectional Synchronization of Multiple Views of Software Models. In Dirk Fahland, Daniel A. Sadilek, Markus Scheidgen, and Stephan Weißleder, editors, *Proceedings of the Workshop on Domain-Specific Modeling Languages (DSML-2008)*, volume 324 of *CEUR-WS*, pages 7–19, 2008.

Gonzalo Génova. What is a Metamodel: The OMG's Metamodeling Infrastructure. *Software and Systems Modeling*, 4(2):171–188, 2005.

L. George, A. Wider, and M. Scheidgen. Type-Safe Model Transformation Languages as Internal DSLs in Scala. In Zhenjiang Hu and Juan de Lara, editors, *Proceedings of the 5th International Conference on Model Transformation (ICMT'12), Prague, Czech Republic, May 28-29, 2012*, volume 7307 of *LNCS*, pages 160–175. Springer, Heidelberg, 2012.

Lars George. Eine interne DSL für Modelltransformationen in Scala. Master's thesis, Beuth-Hochschule für Technik Berlin, May 2012.

H. Giese and R. Wagner. From Model Transformation to Incremental Bidirectional Model Synchronization. *Software and Systems Modeling*, 8(1):21–43, 2009. ISSN 1619-1366.

Ulrike Golas, Leen Lambers, Hartmut Ehrig, and Holger Giese. Toward Bridging the Gap Between Formal Foundations and Current Practice for Triple Graph Grammars. In *Graph Transformations*, pages 141–155. Springer, 2012.

S. Günther and T. Cleenewerck. Design principles for internal domain-specific languages: A pattern catalog illustrated by ruby. In *Proceedings of the 17th Conference On Pattern Languages Of Programs (PLOP 2010), Reno/Tahoe, Nevada, USA*, volume 3, pages 1–2, 2010.

M.H. Halstead. *Elements of Software Science*. Elsevier Science Inc., 1977.

Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Model-Based Language Engineering with EMFText. In *Generative and Transformational Techniques in Software Engineering IV*, pages 322–345. Springer, 2013.

Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, and Yingfei Xiong. Correctness of Model Synchronization Based on Triple Graph Grammars. In *MoDELS*, pages 668–682, 2011.

Frank Hermann, Hartmut Ehrig, Claudia Ermel, and Fernando Orejas. Concurrent Model Synchronization With Conflict Resolution Based on Triple Graph Grammars. *Fundamental Approaches to Software Engineering*, pages 178–193, 2012.

Anders Hessellund, Krzysztof Czarnecki, and Andrzej Wasowski. Guided Development with Multiple Domain-Specific Languages. In *MoDELS*, pages 46–60, 2007.

Anders Hesselund. *Domain-specific Multimodeling.* PhD thesis, IT University of Copenhagen, 2009.

T. Hettel, M. Lawley, and K. Raymond. Model Synchronisation: Definitions for Round-Trip Engineering. *Theory and Practice of Model Transformations*, pages 31–45, 2008.

S. Hidaka, Z. Hu, K. Inaba, H. Kato, and K. Nakano. GRoundTram: An Integrated Framework for Developing Well-behaved Bidirectional Model Transformations. In *26th IEEE/ACM International Conference On Automated Software Engineering (ASE 2011), Oread, Lawrence, Kansas, USA*, pages 480–483. IEEE, 2011.

Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic Embedding of DSLs. In *7th Conference on Generative Programming and Component Engineering*, GPCE '08, pages 137–148. ACM, 2008. ISBN 978-1-60558-267-2.

Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Symmetric Lenses. In *ACM SIGPLAN Notices*, volume 46, pages 371–384. ACM, 2011.

John E Hopcroft. *Introduction to Automata Theory, Languages, and Computation.* Pearson Education, 1979.

F. Jouault and I. Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2006.

Frédéric Jouault and Jean Bézivin. KM3: A DSL for Metamodel Specification. In *Formal Methods for Open Object-Based Distributed Systems*, pages 171–185. Springer, 2006.

Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A Model Transformation Tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

Elina Kalnina and Audris Kalnins. DSL Tool Development with Transformations and Static Mappings. In *Models in Software Engineering, Workshops and Symposia at MODELS 2008. Reports and Revised Selected Papers*, volume 5421 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2008.

Lennart CL Kats and Eelco Visser. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In *ACM Sigplan Notices*, volume 45, pages 444–463. ACM, 2010.

Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly Typed Heterogeneous Collections. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 96–107. ACM Press, 2004. ISBN 1-58113-850-4.

Anneke Kleppe. A Language Description is More Than a Metamodel. In *Proceedings of the 4th International Workshop on Software Language Engineering, Nashville, USA, October 2007.*, 2007.

Filip Křikava and Philippe Collet. On the Use of an Internal DSL for Enriching EMF Models. In *Proceedings of the 12th Workshop on OCL and Textual Modelling*, pages

25–30. ACM, 2012.

Filip Křikava, Philippe Collet, Robert France, et al. SIGMA: Scala Internal Domain-Specific Languages for Model Manipulations. In *MODELS-17th International Conference on Model Driven Engineering Languages and Systems*, 2014.

Ivan Kurtev, Jean Bézivin, and Mehmet Aksit. Technological Spaces: An Initial Appraisal. In *CoopIS, DOA'2002 Federated Conferences, Industrial Track*, 2002.

Ralf Lämmel and Simon Peyton Jones. Scrap Your Boilerplate With Class: Extensible Generic Functions. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 204–215, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7.

Peter J Landin. The Next 700 Programming Languages. *Communications of the ACM*, 9(3):157–166, 1966.

Erhan Leblebici, Anthony Anjorin, Andy Schürr, Stephan Hildebrandt, Jan Rieke, and Joel Greenyer. A Comparison of Incremental Triple Graph Grammar Tools. In *Proceedings of the 13th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2014)*, volume 67. EASST, 2014.

K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization Transformation Based on Automatic Derivation of View Complement Functions. In *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, page 58. ACM, 2007.

Marjan Mernik, Jan Heering, and Anthony M Sloane. When and How to Develop Domain-Specific Languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

Robert N Moll, Michael A Arbib, and Assaf J Kfoury. *An Introduction to Formal Language Theory*. Springer, 1988.

Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, pages 117–120. ACM, 2012.

Jan P Nytun, Andreas Prinz, and Merete S Tveit. Automatic Generation of Modelling Tools. In *Model Driven Architecture–Foundations and Applications*, pages 268–283. Springer, 2006.

OMG. UML 2.0, Infrastructure Specification. *OMG, Needham*, 2004.

Fernando Orejas, Artur Boronat, Hartmut Ehrig, Frank Hermann, and Hanna Schölzel. On Propagation-Based Concurrent Model Synchronization. *Electronic Communications of the EASST*, 57, 2013.

Hugo Pacheco and Alcino Cunha. Generic Point-Free Lenses. In *Proceedings of the 10th International Conference on Mathematics of Program Construction (MPC'10), Québec City, Canada, June 21-23, 2010*, volume 6120 of *LNCS*, pages 331–352, Berlin, Heidelberg, 2010. Springer. ISBN 3-642-13320-7, 978-3-642-13320-6.

C. Picard. Model Transformation With Scala. Master's thesis, Universitat Politècnica de Catalunya, 2008.

Reinhard Pointner. An Evaluation of Scala as a Host Language for DSLs. Technical report, University of Central Lancashire, June 2010.

Andreas Prinz, Robert Eschbach, and Reinhard Gotzhein. A Executable Formal Semantics for SDL-2000. In *SAM'00. 2nd Workshop on SDL and MSC, Col de Porte, Grenoble, France*, pages 249–261, 2000.

Daniel A. Sadilek. *Test-Driven Language Modeling.* PhD thesis, Humboldt-Universität zu Berlin, 2011.

Isao Sasano, Zhenjiang Hu, Soichiro Hidaka, Kazuhiro Inaba, Hiroyuki Kato, and Keisuke Nakano. Toward Bidirectionalization of ATL With GRoundTram. In Jordi Cabot and Eelco Visser, editors, *Theory and Practice of Model Transformations, Fourth International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*, volume 6707 of *Lecture Notes in Computer Science*. Springer, 2011.

Markus Scheidgen. *Description of Computer Languages Based on Object-Oriented Metamodelling.* PhD thesis, Humboldt-Universität zu Berlin, October 2008.

D.C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39 (2):25–31, 2006.

Martin Schmidt. Einsatz modellgetriebener Entwicklung im Bereich der Experimentalphysik. Master's thesis, Beuth Hochschule für Technik, 2011.

Martin Schmidt, Arif Wider, Markus Scheidgen, Joachim Fischer, and Sebastian von Klinski. Refactorings in Language Development with Asymmetric Bidirectional Model Transformations. In Ferhat Khendek, Maria Toeroe, Abdelouahed Gherbi, and Rick Reed, editors, *SDL 2013: Model-Driven Dependability Engineering - 16th International SDL Forum, Montreal, Canada, June 26-28, 2013. Proceedings*, volume 7916 of *Lecture Notes in Computer Science*, pages 222–238. Springer, 2013. ISBN 978-3-642-38910-8.

Andy Schürr and Felix Klar. 15 Years of Triple Graph Grammars. In *ICGT*, pages 411–425, 2008.

Fredrik Seehusen and Ketil Stølen. An Evaluation of the Graphical Modeling Framework (GMF) Based on the Development of the Coras Tool. In *Theory and Practice of Model Transformations*, pages 152–166. Springer, 2011.

Anthony M. Sloane. Experiences with Domain-Specific Language Embedding in Scala. In Julia Lawall and Laurent Réveillère, editors, *Proceedings of the 2nd International Workshop on Domain-Specific Program Development*, 2008.

D. Spiewak and T. Zhao. ScalaQL: Language-Integrated Database Queries for Scala. In *2nd Conference on Software Language Engineering (SLE'09)*, pages 154–163. Springer, 2010.

Diomidis Spinellis. Notable Design Patterns for Domain-Specific Languages. *Journal of*

*Systems and Software*, 56(1):91–99, 2001.

O. Spjuth, T. Helmus, E.L. Willighagen, S. Kuhn, M. Eklund, J. Wagener, P. Murray-Rust, C. Steinbeck, and J.E.S. Wikberg. Bioclipse: An Open Source Workbench for Chemo- and Bioinformatics. *BMC Bioinformatics*, 8(1):59, 2007.

Herbert Stachowiak. *Allgemeine Modelltheorie [General Model Theory]*. Springer, 1973. ISBN 32-1181-106-0.

Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: Eclipse Modeling Framework – Second Edition*. Pearson Education, 2008.

P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Proc. of the 10 Int. Conf. on Model Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, pages 1–14. Springer, 2007a.

P. Stevens. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. *Software and Systems Modeling*, 9(1):7–20, 2010. ISSN 1619-1366.

Perdita Stevens. A Landscape of Bidirectional Model Transformations. In *GTTSE*, pages 408–424, 2007b.

Perdita Stevens. Towards an Algebraic Theory of Bidirectional Transformations. In *ICGT*, pages 1–17, 2008.

P. Suppes. A Comparison of the Meaning and Uses of Models in Mathematics and the Empirical Sciences. *Synthese*, 12(2):287–301, 1960.

C. Szyperski. Components vs. Objects vs. Component Objects. In *Proceedings of OOP '99*, 1999.

Juha-Pekka Tolvanen and Steven Kelly. MetaEdit+: Defining and Using Integrated Domain-Specific Modeling Languages. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 819–820. ACM, 2009.

Mark van den Brand, H. A. de Jong, Paul Klint, and Pieter A. Olivier. Efficient annotated terms. *Software – Practice and Experience*, 30(3):259–291, 2000.

Oskar van Rest, Guido Wachsmuth, Jim RH Steel, Jörn Guy Süß, and Eelco Visser. Robust Real-Time Synchronization Between Textual and Graphical Editors. In *Theory and Practice of Model Transformations*, pages 92–107. Springer, 2013.

Markus Vöelter, Daniel Ratiu, Bernd Kolb, and Bernhard Schaetz. mbeddr: Instantiating a Language Workbench in the Embedded Software Domain. *Automated Software Engineering*, 2013.

M Völter, E Visser, S Kelly, A Hulshout, J Warmer, PJ Molina, B Merkle, and K Thoms. Language Workbench Competition (2011). In *Code Generation Conference*, 2011.

Markus Völter and Konstantin Solomatov. Language Modularization and Composition with Projectional Language Workbenches Illustrated With MPS. *Software Language*

*Engineering, SLE*, page 16, 2010.

Markus Völter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart CL Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering – Designing, Implementing and Using Domain-Specific Languages*. dslbook. org, 2013.

A. Wider. Towards Combinators for Bidirectional Model Transformations in Scala. In Anthony Sloane and Uwe Assmann, editors, *Post-Proceedings of the 4th International Conference on Software Language Engineering (SLE'11), Braga, Portugal, July 3-4, 2011*, volume 6940 of *LNCS*, pages 367–377. Springer, Heidelberg, 2012.

Arif Wider. Towards Lenses for View Synchronization in Metamodel-Based Domain-Specific Workbenches. In *3rd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe '11) at conference INFORMATIK 2011, Berlin, Germany*, GI-Edition of Lecture Notes in Informatics (LNI). Bonner Köllen Verlag, 2011.

Arif Wider. Implementing a Bidirectional Model Transformation Language as an Internal DSL in Scala. In K. Selçuk Candan, Sihem Amer-Yahia, Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proceedings of the Third International Workshop on Bidirectional Transformation (BX'14), co-located with EDBT/ICDT 2014 Joint Conference, Athens, Greece, March 28, 2014*, volume 1133 of *CEUR Workshop Proceedings*, pages 63–70. CEUR-WS.org, 2014.

Arif Wider, Martin Schmidt, Frank Kühnlenz, and Joachim Fischer. A Model-Driven Workbench for Simulation-Based Development of Optical Nanostructures. In *Proceedings of the 2nd International Conference on Computer Modelling and Simulation (CSSim'11), Brno, Czech Republic, September 5-7, 2011*, pages 187 – 195, Brno, Czech Republic, 09 2011. Brno University of Technology (BUT). ISBN 978-80-214-4320-4.

John Wilson-Kanamori and Soichiro Hidaka. A Bidirectional Collaboration Framework for Bio-Model Development. *2nd International Workshop on Bidirectional Transformations (BX 2013), Rome, Italy, colocated with ETAPS 2013, March 17, 2013.*, 2013.

M. Wimmer and G. Kramler. Bridging Grammarware and Modelware. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 159–168. Springer, 2005.

Yingfei Xiong, Hui Song, Zhenjiang Hu, and Masato Takeichi. Supporting Parallel Updates with Bidirectional Model Transformations. In *ICMT '09: Proceedings of the Second International Conference on Theory and Practice of Model Transformations*, pages 213–228, New York, NY, USA, June 2009. Springer. ISBN 978-3-642-02407-8.

# List of Figures

# Definition Index

# List of Abbreviations

API Application programming interface, page 99

ATL Atlas transformation language, page 5

DSL Domain-specific language, page 1

EBNF Extended Backus-Naur form, page 48

EMF Eclipse modeling framework, page 5

FDTD Finite differences time domain method, page 45

GPL General-purpose (programming) language, page 28

GUI Graphical user interface, page 45

HList Heterogeneously typed list, page 116

IDE Integrated (software) development environment, page 1

JDT Eclipse Java development tools, page 2

JVM Java virtual machine, page 39

M2C Model-to-code transformation, page 17

M2M Model-to-model transformation, page 17

M2T Model-to-text transformation, page 17

MDA Model-driven architecture, page 14

MDE Model-driven engineering, page 1

MOF Meta-object facility, page 14

MPS Meta-programming system, page 77

OCL Object constraint language, page 23

OMG Object management group, page 14

QVT Queries, Views, Transformations, page 5

SLE Software language engineering, page 1